1996

Arbitrary views of high-dimensional space and data

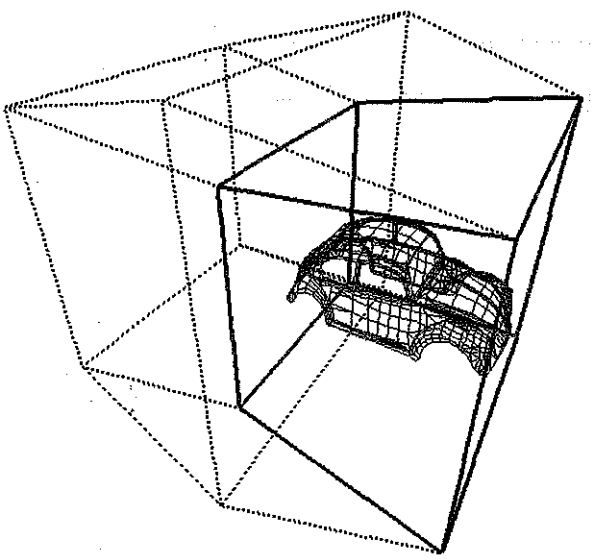
Andrew Ellerton
Edith Cowan University

Edith Cowan University

Copyright Warning

Arbitrary Views of High-Dimensional Space and Data

Andrew Ellerton



USE OF THESIS

The Use of Thesis statement is not included in this version of the thesis.

Arbitrary Views of High-Dimensional Space and Data

Andrew Ellerton

Date of Submission: 31 January 1996

Thesis submitted in partial fulfilment of the Requirements for the Award of Bachelor of Science (Computer Science) Honours, Faculty of Science, Mathematics and Engineering, Edith Cowan University.

Arbitrary Views of High-Dimensional Space and Data

Candidate: Andrew ELLERTON

Student Number: [redacted]

Institution: Edith Cowan University

Award: Bachelor of Science (Computer Science) Honours

Supervisor: Dr Thomas O'Neill

Arbitrary Views of High-Dimensional Space and Data

Abstract

Computer generated images of three-dimensional scenes objects are the result of parallel/perspective projections of the objects onto a two-dimensional plane. The computational techniques may be extended to project n-dimensional hyperobjects onto $(n-1)$ dimensions, for $n > 3$. Projection to one less dimension may be applied recursively for data of any high dimension until that data is two-dimensional, when it may be directed to a computer screen or to some other two-dimensional output device.

Arbitrary specification of eye location, target location, field-of-view angles and other parameters provide flexibility, so that data may be viewed—and hence perceived—in previously unavailable ways. However, arbitrary views may also increase the computational requirements, and may complicate the user's task in preparing and interpreting a view.

Data with a dimension greater than three are difficult to perceive geometrically, yet may be invaluable to the observer. This study designs and implements a data visualisation system which incorporates arbitrary views of high-dimensional objects using repeated hyperplanar projection.

Declaration

I certify that this thesis does not incorporate without acknowledgement any material previously submitted for a degree or diploma in any institution of higher education; and that to the best of my knowledge and belief it does not contain any material previously published or written by another person except where due reference is made in the text.

Signature:

Date: 18/2/96

Acknowledgements

Thanks—

To Thomas O'Neill for so many hours spent supervising and guiding me above and beyond this project. To N. F. Ellerton, Derek and my family for their support. To S'Lei, whose patience, interest and encouragement helped me to finish this project. Many thanks also to: Mike Collins, H. T. Lee, Jon Mo, Shelby Oliver, Ho Tang and Jennette Surasathian.

To CSIRO and ACSys at the Australian National University in Canberra, David O'Brian of Curtin University of Technology, Mike Hartley of the University of Western Australia, and to Earth Resource Mapper Inc. of Perth, Western Australia.

To God, above all, for the gifts to be able to pursue this project.

—Andrew Ellerton, January 1996

Contents

List of Figures

List of Tables

1. Introduction

Existing three-dimensional computer graphics algorithms may be generalised to allow projection and subsequent rendering of n-dimensional data on a two-dimensional output device. When used in conjunction with other computer graphics facilities, including real-time animation and depth cues such as shading and hidden surface elimination, an implementation of the generalised algorithm may provide a powerful tool for the visualisation of highly-multivariate or high-dimensional datasets. As a visualisation technique, it offers potential advantage in scientific fields: namely, including mathematics, statistical analysis, geographic information science (GIS) and computer graphics in general.

The chapters of this study present an overall understanding of the background and problems encountered in rendering images of high-dimensional data. This chapter serves a two-fold purpose:

- to provide an introduction to the salient concepts of dimensionality, computer graphics techniques, and data visualisation; and,
- to summarise the purpose and direction of each subsequent chapter.

1.1 Salient Concepts

Measurements obtained through empirical or analytical means may be used to represent a particular theory, a formula, a state, an event or an observation. The collected numerical data might also be used to build a model to enhance understanding of, or better explain, the interrelationships among the data (Beddow, 1990). Various algorithms or analytical processes of some form may be applied to the data, enhancing the ability to see and assess relationships, or to analyse patterns not previously apparent.

The term *visualisation* (or sometimes *scientific visualisation*) may be applied to any process involving the conversion of numerical measurements to an image (Nielson, Shriver & Rosenblum, 1990; van Walsum & Post,

1994). This is supported by van Walsum and Post (1994, p. 339), who state that

> ... visualization is concerned with techniques that transform large scientific datasets into images. The goal of visualization is to provide insight in the dataset; the numbers in the dataset must be transformed to useful information.

Within the dataset, a single datum or tuple may be comprised of one or more singletons, each representing exactly one measurement of some form. The *order* or *dimensionality* of a tuple is the number of measurements involved in its definition.

For example, a position on a grid may be denoted with the two-dimensional tuple $\mathbf{p} = (x, y)^{\mathrm{T}}$, where x and y may be distances along orthogonal axes from an origin (as with Cartesian coordinates), or a scalar distance and an angle from an origin (as with polar coordinates). A third component, colour for example, might be added to the tuple, redefining it as $\mathbf{p} = (x, y, c)^{\mathrm{T}}$. The new tuple would then describe the colour information at any point on a grid; thus, a collection of such tuples would form a tuple-space describing an image on a computer screen.

According to LeBlanc & Ward (1990, p. 230), tuples of order one to four may be termed *low order or low-dimensional*, while tuples of order five or higher may be termed *high order* or *high-dimensional*. However, because anything more than the familiar three spatial dimensions is alien to humans, a tuple of order four and higher is termed high-dimensional throughout this document.

Visualisation of low-dimensional tuples may be considered trivial using any of a number of computer graphics and visualisation techniques including number-lines, scatter-plots, graphs, charts, volume rendering (Clifton & Wefer, 1993) and abstract representations (LeBlanc & Ward, 1990; Mihalisin, Gawlinski, Timlin & Schwegler, 1990).

However, in order to visualise three-dimensional data, the dataset must necessarily be reduced to the dimensionality of the display device. Computer graphics applications frequently use perspective and parallel projection to transform a three-dimensional dataset to the two-dimensional space of a plane, which may then be mapped to an output device. Although

the projected image is two-dimensional, the illusion of a third dimension is achieved through use of one or more depth cues (Clifton & Wefer, 1993, p. 57) which, among others, include:

- perspective foreshortening (more distant objects appear smaller);
- shading, shadowing (colouring an object based on attributes such as position and/or orientation);
- occlusion (closer objects obscure further objects);
- texture gradient (closer objects have greater detail); and,
- motion parallax (animation, such as a changing observer location/ view-point, alters the projected image even though the dataset remains constant).

Although other forms of projection exist, they all "transform points in a coordinate system of dimension n into points in a coordinate system of dimension less than n" (Foley et al., 1991, p. 230). Further, Noll (1967, p. 469) suggests that

> The mathematics and projective geometry of three-dimensional space can be generalized to any number of dimensions so that an n-dimensional hyperobject can be mathematically projected into an $(n-1)$-dimensional space. Such projection could be applied repetitively until finally a three-dimensional object representing the successive projections of an n-dimensional hyperobject is obtained.

Noll's notion of repeated high-dimensional planar projection forms the basis of the multi-dimensional visualisation algorithm developed in this study.

As Noll's method is a n-dimensional generalisation of three-dimensional planar projection (a principal component of three-dimensional computer graphics), other aspects of three-dimensional computer graphics techniques may also be generalised and utilised in the high-dimensional visualisation system. For example, planar projection requires the specification of, at least, a view-point in the three-dimensional space, and the position and orientation of a plane on which to project the scene. The view-point and plane position/orientation are examples of *viewing parameters*. These combined with other parameters yet to be discussed,

define a *view volume* within three-dimensional space which will be projected onto the plane (Foley et al., 1991, p. 229).

Implementing three-dimensional planar projection is mathematically convenient only under certain conditions (Foley et al., 1991, p. 253; Noll, 1967, p. 470): specifically, when

- the view-point is positioned along the z-axis;
- the target-point and the projection plane (centred at the target-point) are positioned along the z-axis; and,
- the projection plane is defined as normal to the z-axis.

In practice, however, these limitations are unrealistic and forbidding because a "real" image may require viewing from any arbitrary point in space (or perhaps a changing sequence of points in the case of animation) and with an arbitrarily placed and oriented projection plane (Foley et al., 1991, p. 237). The incorporation of other viewing parameters, such as viewport specification, clipping planes, an "up" direction and field-of-view angle, allows visualisation from arbitrary points in space of any arbitrary volume in the same space (Foley et al., 1991, pp. 239–242).

Foley et al. (1991, p. 238) term a coordinate system that adheres to the projection constraints as the *viewing-reference coordinate* system; in constrast, the *world coordinate* system may be thought of as an unconstrained system where the viewing parameters are arbitrary.

Arbitrary views of a dataset provide great power and flexibility to the user of the visualisation technique (Clifton & Wefer, 1993; van de Grind, 1986). These views may be achieved by transforming the initial arbitrary coordinate system (the world coordinate system in Foley et al.) such that it becomes the trivial projection case (the viewing-reference coordinate system) as detailed above (Foley et al., 1991, pp. 241–242).

Using a sequence of "primitive" transformations to rotate, shape and move the view volume, the mathematically convenient orientation, size and location may be achieved (Foley et al., 1991, pp. 253–281). Such primitive transformations include translation (movement along one or more axes), scaling (enlarging or shrinking along all axes), rotation (rotation about an axis) and shearing (scaling along one or more axes independently of other

axes). These transformations are well established in two- and three-dimensions (Foley et al., 1991; Hearn & Baker, 1994; Hill, 1990; Watt, 1989) and are directly extensible to the geometry of n-dimensions (Noll, 1967; Manning, 1921; Sommerville, 1958).

In summary, the use of a repeated projection algorithm transforms points in n-dimensional space to points in a k-dimensional subspace, where $n > k > 0$. Specification of arbitrary views of high-dimensional space may assist production of meaningful images, and help to facilitate interaction with the projected image. Arbitrary views of high-dimensional space and data may be achieved by generalising the three-dimensional viewing process, including the use of n-dimensional generalisations of three-dimensional transformations.

1.2 Document Structure

The remaining chapters examine the design and implementation of a dimensionally generalised planar projection algorithm which allows n-dimensional coordinates to be projected to k-dimensions ($n > k > 0$), and how n-dimensional coordinate transformations and arbitrary views may be achieved.

Chapter two discusses the background of the study and defines the research questions. The criteria and boundaries for the high-dimensional visualisation to be developed are also set.

Chapter three, through an examination of past and present research, provides a discussion and a critique of existing visualisation and dimensional reduction techniques for high-dimensional, or highly-multivariate data. This literature review qualifies the development of hyperplanar projection through contrast with other comparable high-dimensional visualisation techniques.

Chapter four details the design and implementation of the high-dimensional visualisation system. The mathematics of planar projection and three-dimensional computer graphics are discussed as a background and precursor to the n-dimensional forms. The mathematics and algorithms underlying hyperplanar geometric projection and the n-dimensional viewing process are then elaborated. The chapter concludes

with the definition of, and reasoning behind, the implementation framework.

Chapter five presents the results of the study's implementation in the form of a small gallery of images and example programs. The effectiveness of the projection algorithm for processing arbitrary views of high-dimensional spaces, subspaces and data, is demonstrated. Several examples are provided which demonstrate contrasting images produced from different views of the same data.

Chapter six concludes the project. Assessment is made of the design and implementation with respect to the research questions and criteria previously defined. Areas where further research might be undertaken are outlined, and a list of references is provided.

The appendices concluding the document supply additional detail to support and expand upon topics discussed in the main body of the text. A summary of each follows:

- appendix A provides an overview of the modules in the implementation and maintenance information, as well as listings of the source code, including header files, that comprise the implementation of the design;
- appendix B contains listings of the demonstration programs built using the library; and,
- appendix C contains listings of the test drivers used to verify correctness of the modules detailed in appendix A.

2. Framework

A visualisation system is designed to produce images based on data from some problem domain, such that any useful information about the data may be understood from inspection of the image. In general, the system should fulfil specific requirements within any limitations, and according to any assumptions, as required. In order to establish a framework for the design and implementation presented in later chapters, this chapter examines the background, significance, requirements and limitations of the visualisation system developed by the present study.

2.1 Background to the Study

Scientific visualisation techniques may allow patterns, relationships, clusters and outliers in a dataset to be ascertained. Visualisations are limited to the two- or three-dimensions of the output device, yet high-dimensional datasets are frequently encountered in a variety of fields, including statistics (Crawford & Fall, 1990), mathematics (Banchoff, 1990; Cole, 1993) and geographic applications (Lindgren, 1978). Such datasets cannot be visualised directly because the output devices are limited dimensionally.

In order to visualise high-dimensional data on these limited devices, the dimensionality of the dataset must necessarily be reduced. Various *dimensional reduction techniques* exist, each with particular strengths, weaknesses and applications. For example, an *abstract representation* allows a number of dimensions to be represented by encoding each one in some manner—Chernoff (in Crawford & Fall, 1990) maps each of twelve dimensions to a facial characteristic, and renders one face per tuple. Thus, a dataset is represented by a range of faces—and those faces with large smiles, for example, might have a large positive value in one of the dimensions.

Another dimensional reduction method is *projection*. For example, a three-dimensional volume may be projected to a two-dimensional surface. *Three-dimensional planar projection* is where points in three-dimensional

space are projected to a two-dimensional plane, rather than a curved surface (Foley et al., 1991, p. 230). Real-life images such as mirror reflections, maps and photographs result from three-dimensional planar projection. Further, planar projection is used in many three-dimensional computer graphics applications, ranging from games to statistical applications and real-time simulations.

The mathematical extension of three-dimensional planar projection to higher-dimensions, such as that used in computer graphics, appears straightforward. This *high-dimensional hyperplanar projection* might be used to reduce a n-dimensional space to a $(n-1)$-dimensional space. Such a projection may be applied repetitively, reducing the dimensionality of a dataset to any positive number.

Given a n-dimensional dataset, where $n > 3$, a two-dimensional image of that dataset may be rendered on a computer output device after applying hyperplanar projection $(n-2)$ times. That is, a four-dimensional dataset must be projected twice—firstly, from a four-dimensional space to a three-dimensional subspace; and, secondly, from the three-dimensional subspace to a two-dimensional subspace.

The rendered image might then be assessed for relationships or patterns in the original dataset. If, prior to projection, the n-dimensional dataset were to be rotated, and positioned at will, the projected image would then represent the same dataset from a different point-of-view. Thus, in the same way that a three-dimensional scene may be rotated and viewed from any point in space, a n-dimensional "scene" may also be viewed from any n-dimensional point.

Further, established three-dimensional computer graphics tools and techniques, including depth cues (such as colouring, depth shading, and hidden surface removal), view specification, interaction and animation, may enhance the high-dimensional visualisation system and increase the amount of useful information that may be read from an image. For example, a certain dataset may look entirely different from various points-of-view—so animating the image by moving the observer's position in space may enhance the viewer's perception of the construction of the scene. Such

arbitrary views of a scene may be achieved by incorporating a set of *viewing parameters* into the viewing process.

Examples of viewing parameters include the *view-point* (the observer's position in space), the *target-point* (the focus of the observer) and view-volume boundaries (the volume in space to be projected to the display device).

Interaction with three-dimensional viewing parameters may assist the viewer in understanding the image and original three-dimensional scene. Similarly, user interaction with the parameters controlling the view of a high-dimensional scene may also yield useful information about the original high-dimensional space.

2.2 Significance of the Study

A repeated hyperplanar projection allows an interpretation of a n-dimensional dataset to be displayed on a computer output device. This method may provide a powerful tool for the visualisation and understanding of multi-dimensional datasets.

Further, repeated hyperplanar projection appears both powerful and straightforward. An entire dataset may be projected to a single image, in comparison to some techniques (Crawford & Fall, 1990; McDonald, 1983) which produce a vast array of images, each of which must then be examined. Projection may also be used in conjunction with existing computer graphics techniques, including shading, animation and view specification, to increase the amount of tangible information depicted by the image, thus offering assistance to the viewer in assessing that information.

2.3 Statement of the Requirements

This study investigates a method to display projections of n-dimensional datasets. The method must satisfy two requirements:

- the establishment of hyperplanar projection of a n-dimensional space to a $(n-1)$-dimensional space; and,
- the enablement of repeated hyperplanar projection, such that a n-dimensional space may be projected to a k-dimensional space, for some k where $n > k > 0$.

Further, the visualisation system should take advantage of existing computer graphics facilities and visualisation aids; hence, additional requirements are:

- the incorporation of viewing parameters and n-dimensional transformations to provide arbitrary views of high-dimensional datasets;
- the employment of computer graphics techniques, such as colour shading, hidden line/surface removal and animation; and,
- user interaction with the visualisation tools, in order to allow modification of the projected image and/or the view parameters controlling the production of the image.

2.4 Research Questions to be Answered

The study will focus on the following question:

Given that three-dimensional planar projection may be generalised to n-dimensional hyperplanar projection and three-dimensional coordinate transformations may also be generalised to n-dimensions, how may both be integrated into a high-dimensional viewing process, allowing arbitrary views of a n-dimensional dataset to be projected to any lower dimension and viewed on a computer output device?

Several more explicit questions arise from this postulation:

1. How is hyperplanar projection defined?
2. How may hyperplanar projection be repeated, thereby allowing projection of any n-dimensional space to a k-dimensional subspace, for some k where $n > k > 0$?
3. Given that three-dimensional viewing parameters, such as view position, target position and view-plane orientation/position enable the rendering of arbitrary views of a three-dimensional dataset, how may those parameters be generalised to n-dimensions?
4. How may the n-dimensional viewing process be defined, incorporating both n-dimensional viewing parameters and repeated hyperplanar projection?

5. What implementation framework allows high-dimensional datasets to be rendered taking advantage of existing computer graphics techniques and tools?

2.5 Assumptions and Limitations of the Study

Computer techniques for rendering an arbitrary view of a space are building blocks for more advanced visualisation tools. Such tools might incorporate powerful interface and analysis facilities as controllers/ assistants for understanding and interacting with the data being viewed.

It is important to clarify that this study is endeavouring to investigate only transformations and projection of n-dimensional datasets. The datasets are domain independent — the specific problem domain from which the data were derived or obtained, such as a satellite observation or a statistical source, is of no consequence to the high-dimensional visualisation algorithm developed here.

However, the implementation is intended to be modular, allowing larger, more powerful and complex applications to be built using the high-dimensional visualisation system developed for this study. Relevant functionality and analysis tools might be built into such a visualisation program, which might process and present the data in a manner suitable for a specific problem domain.

2.6 Summary

Visualisation of a dataset with an arbitrarily high number of dimensions may be achieved by repeatedly projecting that dataset to one lower dimension, until the dataset may be output to a device such as a computer monitor. In order to assist interpretation of the rendered image, several facilities should be incorporated into the visualisation system: namely, depth cues, arbitrary views, and interaction with the view parameters (resulting in a real-time animation of the high-dimensional scene).

This chapter has provided a framework for the study, and established the requirements and research questions that the study will answer in the remaining chapters.

3. Literature Review

Through a review of relevant literature, this chapter will establish the foundation for the project by discussing data visualisation in general, and high-dimensional concepts and applications and existing computer techniques for the visualisation of high-dimensional datasets, in particular.

3.1 General Literature

Data visualisation relies upon several areas in computer science. In essence, it is the process of converting numbers into pictures—a form which may facilitate human recognition of possible relationships which might otherwise be difficult to detect from the raw data. As Beddow (1990, p. 238) states:

> the visual representation of data from complex systems, whether databases, measured scientific data, or simulation output, holds the promise of discovering patterns in the data that will increase its management efficiency while revealing relationships invisible to numeric methods.

Haber and McNabb (1990, p. 74) observe that "computer visualization methods have emerged as the most effective tool for rapidly communicating large amounts of information ... in a format that enhances comprehension and deepens insight." Kaufman, Nielson and Rosenblum (1993, p. 16) add that "visualization technology ... has already revolutionized the way scientists do science, the way engineers design, and the way physicians deliver health care."

Visualisations displayed using computer graphics output devices are two-dimensional and, therefore, may only display images of one- and two-dimensional data directly. However, some hardware and software, such as *GL*, *OpenGL*, *Open Inventor*, *PEX*, *PHIGS* and *Quickdraw3D* (Neider, Davis & Woo, 1993; Segal & Akeley, 1994; Wernecke, 1994), provide three-dimensional computer graphics "virtual machines." These allow visualisation of three-dimensional scenes or data to be achieved by

perspective/parallel projection and a combination of depth cues to simulate the third dimension[1].

Unfortunately, datasets are rarely confined to the limiting two- or three-dimensions of the output device (Crawford & Fall, 1990). The quest for *high-dimensional data visualisation*, then, is to find methods of visualising n-dimensional datasets (where $n > 3$), within the limited dimensionality of the output device, while still providing meaningful, accurate and "readable" images. Toward this goal, a *dimensional reduction technique* may be employed to reduce the dimensionality of a dataset or tuple-space to a more manageable level. Various approaches to the reduction of dataset dimensionality include linear mapping, projection, slicing, shadow casting, subspace projection, abstract representation, glyph or icon representation and others; and, discussion of these is deferred to the next section.

Why then is there such an interest in high-dimensional data, geometry and visualisation? Cole (1993, p. 54) has addressed this question:

> Why would mathematicians want to leave the comfort of our familiar three-dimensional world? Because, curiously, by poking their heads up into higher dimensions, they can get a clearer view of complex problems—they can see relationships that look hopelessly tangled in the squashed and compacted universe of lower dimensions.

According to Cole (1993, p. 54), there exists a wide range of potentially important applications of high-dimensional geometry and visualisation:

> ... astrophysicists enter higher dimensions to see patterns in star clusters; particle physicists to look for unified theories; engineers to analyze mechanical linkages; and communications specialists to find ways to pack information into tight spaces. There's nothing like hopping into a higher dimension to make a complex problem easier.

Based on the literature, discernible potential applications of high-dimensional visualisation include, but are not limited to, the following six

---

1. Notably, recent developments with direct volume display devices (DVDDs) may ultimately lead to output devices which display "3D volumes and surfaces in a volume, providing depth rather than depth cues" (Clifton & Wefer, 1993, p. 57).

areas: statistics, mathematics, physics, engineering/computer science, geographic applications and art. Brief comments on applications, in each of these areas, are now provided.

Statistics and quantitative research. Much of the research into high-dimensional visualisation has been directed toward statistical applications, including work by Crawford and Fall (1990), who, among others, continued some of the pioneering statistical visualisation research of Fisherkeller, Friedman and Tukey (1974). Crawford and Fall (1990, p. 94) point out that "modern data collection is now so efficient that researchers in many fields must analyze very large quantities of highly multivariate data." High-dimensional visualisation seeks the "uncovering [of] structure" (Crawford & Fall, 1990, p. 94), including "such phenomena as local extrema, trends, discontinuities, anomalies and correlations" (LeBlanc, Ward & Wittels, 1990, p. 231).

Mathematics. Although measurement and dimensions are essentially mathematical, some applications are "purely" mathematical. Notably, Cole (1993) states that mathematicians "travel routinely not only to the fifth dimension but also to the seventh, the tenth and the twenty-sixth." Specific areas include:

- Visualisation of functions of more than three variables, such as quaternions (Banchoff, 1990, p. 175; Hearn & Baker, 1994, p. 384) and fractals (Hearn & Baker, 1994, p. 385; Norton, 1982);
- Geometry (Banchoff, 1990; Cole, 1993; Koçak, Bissopp, Laidlaw & Banchoff, 1986; Noll, 1967). According to Cole (1993, p. 59), a recent publication incorporated "a method of figuring out the number of possible solutions to complex problems based on the geometry of higher-dimensional spaces."

Physics. Cole (1993, p. 62) notes that some physicists believe that apparently irreconcilable differences between gravity and other fundamental forces "will disappear if we learn the right way to view them in higher dimensions." Specific applications in the realm of physics include:

- Theory of relativity/Visualisation of Einstein's four-dimensional curved space (Cole, 1993; Hawking, 1988, p. 32);
- Movement of planetary bodies and paths of light rays as geodesic in curved four-space (Hawking, p. 33);
- Motion of both molecules and stars is simplified when imagined in higher-dimensional "phase spaces" (Cole, 1993, p. 60);
- String theory (Cole); and
- Astronomy. Cole (1993, p. 56) discusses the case of an astrophysicist who, after working with six-dimensional spaces and looking at globular clusters by analysing the geometry of 60,000-dimensional cigars, exclaimed "it's extraordinary how all these complicated motions can reduce to, say, a simple 67-dimensional doughnut."

Engineering and computer science. Various engineering applications, including mechanical linkage examination (Cole, 1993) have been proposed. In addition, visual programming, visualisation of multiple inheritance relationships, and software analysis may also be potential areas for research.

Geographic applications. During the initial phases of the present project, the author discussed high-dimensional visualisation with several local remote sensing companies, and with the Geographic Information Science (GIS) department of the Curtin University of Technology. All parties contacted believe the results of the research may have applications in their domain of work. Specific areas of interest include:

- Urban and regional planning (Lindgren, 1978); and,
- Quantitative research using visualisation of multivariate data obtained from land surveying (Calder, 1991).

Art. Slaby (1978, p. 19) argues that "some of the most interesting work in multi-dimensional geometry ... is being done by artists," while Cox (1988, p. 233) used data from "such fields as agricultural entomology, topology and astrophysics to render visual representations of

multidimensional computations." Cox also notes that, with respect to high-dimensional visualisation, artist and scientist may "work toward a common goal: visualisation of the impossible." To this end, Norton (1982) produced images of four-dimensional fractals, projected to three-space.

Summary of general literature. In order to support the wide potential areas noted above, it can be seen from the work of others such as Beddow (1990), Crawford & Fall (1990), Donoho, Huber, Ramos & Thoma (1982), Hausmann & Seidel (1994), LeBlanc, Ward & Wittels (1990), Lindgren (1978), McDonald (1983) and Mihalisin, Gawlinski, Timlin & Schwegler (1990) that there has been considerable recent research into the geometry and visualisation of high-dimensional space and data. Specific approaches, advantages and disadvantages of various visualisation methods are discussed in the following section.

3.2 Literature of Particular Relevance to the Current Study

A dataset of dimension n must be reduced to k dimensions in order to display a visual representation of that dataset on a k-dimensional output device. In the case of a computer screen or paper, for example, k may equal one or two. When a three-dimensional output device or virtual machine is used (as discussed in Section 3.1), k may be one, two or three.[2]

In general, however, if $n > 2$, rendering an image of a n-dimensional space requires a dimensional reduction technique to reduce the dimensionality of the dataset. The particular dimensional reduction technique used depends upon the domain and the implementation. That is to say, a particular problem domain may lend itself toward graphs of a particular style—scatter plots for statistics (see, for example, McDonald, 1983), or surfaces for geographic applications (Calder, 1991) and/or mathematical applications (Wolfram, 1991).

Dimensional reduction techniques, generally, may be classified into one of two categories (Crawford & Fall, 1990): A *linear mapping*, or a *non-*

2. Note, however, that although the *OpenGL* graphics library allows k to equal four (Neider, Davis & Woo, 1993, p. 33), the high-dimensional functionality is otherwise limited.

linear mapping/abstract representation. Discussion of each class in more detail may assist comprehension of the various approaches available, and aid understanding of the approach chosen for this study.

Linear Mapping. One or more dimensions of a n-dimensional space may be linearly mapped to one or more of k-dimensions ($k < n$). That is to say, if the n-dimensional tuple $\mathbf{x}^{(n)} = (x_1, x_2, \ldots, x_n)$ is to be reduced to the k-dimensional tuple $\mathbf{x}^{(k)} = (x_1', x_2', \ldots x_k')$, each component x_i' is determined by some function of the form $x_i' = m_i x_i + c$, where m_i is a multiplier and c is a constant offset. Categories of linear dimensional reduction techniques include projection, slicing, shadows and subspace projection.

Projection. A projection of a three-dimensional space may be considered as follows: An object **H** exists within a three-dimensional space **S**, and is defined, for simplicity, by a number of connected points. Within **S**, a two-dimensional projective space **S'** is placed between the object and a view-point **v** in space. For every vertex **x** defining **H**, a *projector* from **v** to **x** is calculated, intersecting **S'** at the point **x'**. This point of intersection is the *projection of* **x** onto **S'**. The set of line segments connecting the intersection points is the projection of the object **H** onto **S'**.

Projections are commonly categorised by the shape of the projective space (planar or curved, for example), the form of the projectors (straight lines or curved lines), and other parameters such as the position and orientation of the projective space and the view point. However, as noted in chapter one, all forms of projection "transform points in a coordinate system of dimension n into points in a coordinate system of dimension less than n" (Foley et al., 1991, p. 230). This statement reiterates a well-known mathematical observation; furthermore, on the subject of repeated use of projection, Noll (1967, p. 469) suggests that

> the mathematics and projective geometry of three-dimensional space can be generalized to any number of dimensions so that an n-dimensional hyperobject can be mathematically projected into an $(n-1)$-dimensional space. Such projection could be applied repetitively until finally a three-

dimensional object representing the successive projections of an n-dimensional hyperobject is obtained.

Use of projection does not discount essential information, rather it may hide/distort some detail; hence, repeated hyperplanar is the approach adopted by this study to reduce the dimensionality of a dataset to any lower dimension.

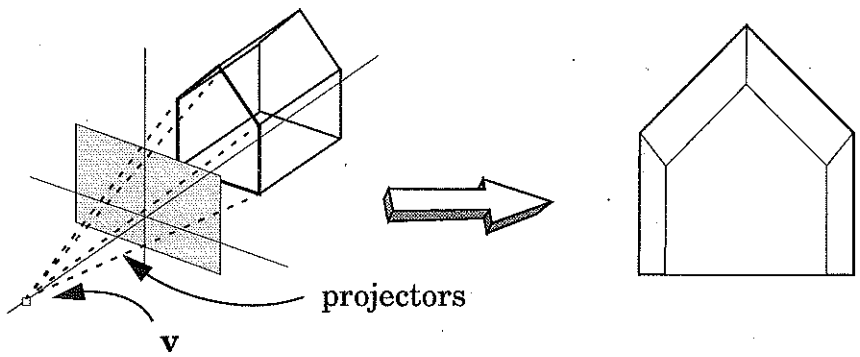
Although discussion of planar projection and hyperplanar projection continues in chapter four, a brief elaboration on two forms of planar projection provides clarification for further discussion.

Perspective projection and *parallel projection* are examples of three-dimensional geometric planar projection.[3] The two forms may be distinguished by the distance from the projection plane to the view-point: perspective projection requires that this distance be finite, whereas parallel projection requires the distance to be infinite (Foley et al., 1991, p. 230). The visual distinction between the two is that a perspective image exhibits non-uniform foreshortening (more distant objects appear smaller than closer objects), which is useful for realistic rendering of scenes. Conversely, parallel projections exhibit either no foreshortening (more distant objects appear the same size as closer objects) or foreshortening is uniform (so distances and sometimes angles may be measured from an image). Parallel projections are useful for design-oriented applications including computer-aided design. Perspective projections are considered more realistic, but angles and distances cannot generally be read from the image. Examples of perspective and parallel projections are shown in Figure 3.1.

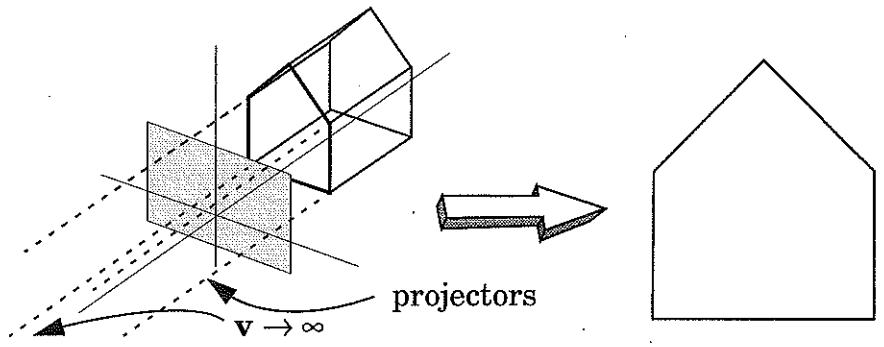
Various forms and applications of parallel and perspective projections are comprehensively reviewed by Carlbom and Paciorek (1978). Moreover, although unsuitable for employment in dynamic image rendering, there exist non-geometric and non-planar projections which have extensive applications including wide-format film and cartography (Foley et al., 1991, p. 230).

---

3. Geometric planar projection: "geometric" means that the projectors are straight lines, "planar" means that the projective space is a plane (Foley et al., 1991, p. 230).



(a) Perspective projection setup (left) and resulting image (right)



(b) Parallel projection setup (left) and resulting image (right)

Figure 3.1 Examples of Projection.

Slices. Three-dimensional slicing involves the "extraction of a planar image of arbitrary orientation at a particular location in a 3D data set" (Nelson & Elvins, 1993, p. 52). But slicing does not need to be limited to three-dimensions, for as Cole (1993, p. 57) notes, "just as you might cut a three-dimensional block of cheese into two dimensional slices, so you can cut a four-dimensional block into three-dimensional slices."

Slicing may be considered as similar to projection with the projection space defined at a position and orientation somewhere interior to the object. However, slicing is different because the "slice" obtained includes only data within, and necessarily excludes data from behind or in front of the projection space. For example, a horizontal slice near the top of a cup might yield a circle—no handle or base would be visible.

By moving or re-orienting the projection space, views of different portions of the dataset can be viewed. Although slicing would appear to have various potential applications, including medical imaging such as

ultrasound scans, these have not yet been greatly explored by research.

Shadow casting. This is also similar to projection, except that the projective space is strictly "behind" the object, as seen from the view point. Projectors are cast from the view point, through the object, and onto the projective space, effectively producing a shadow of the object or dataset. Extrapolating to higher-dimensions Cole (1993, p. 57) notes how "three-dimensional shadows of four-dimensional objects" may be rendered on a computer screen.

Regardless of the dimensionality of an object, its shadow may be a vague representation in the projection subspace. Thus, use of shadows for viewing high-dimensional hyperobjects/datasets is deemed to be of limited demonstration value (Cole, 1993).

Subspace projection. Any two- or three-dimensional mapping may be picked from the total number of dimensions defining the dataset and plotted by traditional means. This approach was taken by Fisherkeller, Friedman and Tukey (1974) with the *PRIM-9* system, and has spawned several descendants, *PRIM-S* and *PRIM-H* (Donoho, Huber, Ramos & Thoma, 1982) and *Orion* (McDonald, 1983).

Potentially, a huge number of images may be generated using subspace projection—Crawford and Fall (1990, p. 94) acknowledge that "sheer combinatorics" limit how many dimensions of a dataset can be displayed in, say, multiple two-dimensional plots. For this reason, there is a need for additional mechanisms to control or choose from a range of images, when working with subspaces of a larger dataset.

One such mechanism is *projection pursuit* (McDonald, 1983; Crawford & Wall, 1990), which is the process of extracting salient (rather than just any) views of a high-dimensional dataset. Crawford and Fall (1990, p. 100) note that automated projection pursuit is basically "an algorithm for heuristic exploration of the multidimensional space." However, although projection pursuit (automatic or otherwise) attempts to remove non-vital views, subspace projection is unattractive because the viewer is not given a detailed insight into the nature of a dataset, and only a small subset of the

dataset (or several subsets) may be viewed at any time.

Various investigations have explored the use of subspaces, combining the visualisation approach with various analytical tools and processes designed to allow visualisation of the entire dataset.

An **abstract representation** of a high-dimensional dataset is one where the dimensionality of the original dataset has been reduced by encoding one or more dimensions in some abstract form. An abstract visualisation of a high-dimensional dataset may have little or no apparent resemblance to the original form of the dataset. This contrasts with linearly reduced visualisations, which tend to exhibit a similar form and/or distribution of the numeric content of the dataset. An abstract representation of a twelve-dimensional dataset is shown in Figure 3.2, and is discussed below. Abstract approaches include glyphs, pseudo-colour and shape-coding, each of which is also discussed briefly below.

Glyph or **iconic representation.** Figure 3.2 shows an example of an abstract representation developed by Chernoff (in Crawford & Fall, 1990). Each of twelve dimensions of a tuple-space is mapped to a facial characteristic; a single tuple maps to a single face of certain features. Glyph's are non-linear representations; a face $\mathbf{F} = (d, s, ...)$, derived from a tuple $\mathbf{v} = (v_1, v_2, ..., v_n)$ may be defined by rules such as "draw the face with diameter d and with a smile of amplitude s." The definitions of d and s may then be functions of $\mathbf{v}$, such as $d = v_1/\text{Max}(v_1)$.

Another common use of glyphs is sunflower displays, "in which each data point becomes an axis radiating lines of different length" (Crawford & Fall, 1990, p. 95) and trees (LeBlanc, Ward & Wittels, 1990), where the characteristics of the tree are determined by values in each tuple.

Other abstract dimensional reduction techniques. Various other forms of abstract dimensional reduction technique have been put forward. These include colour and/or shape coding (Beddow, 1990; Mihalisin et al., 1990) and dimensional "stacking" (LeBlanc, Ward & Wittels, 1990). These

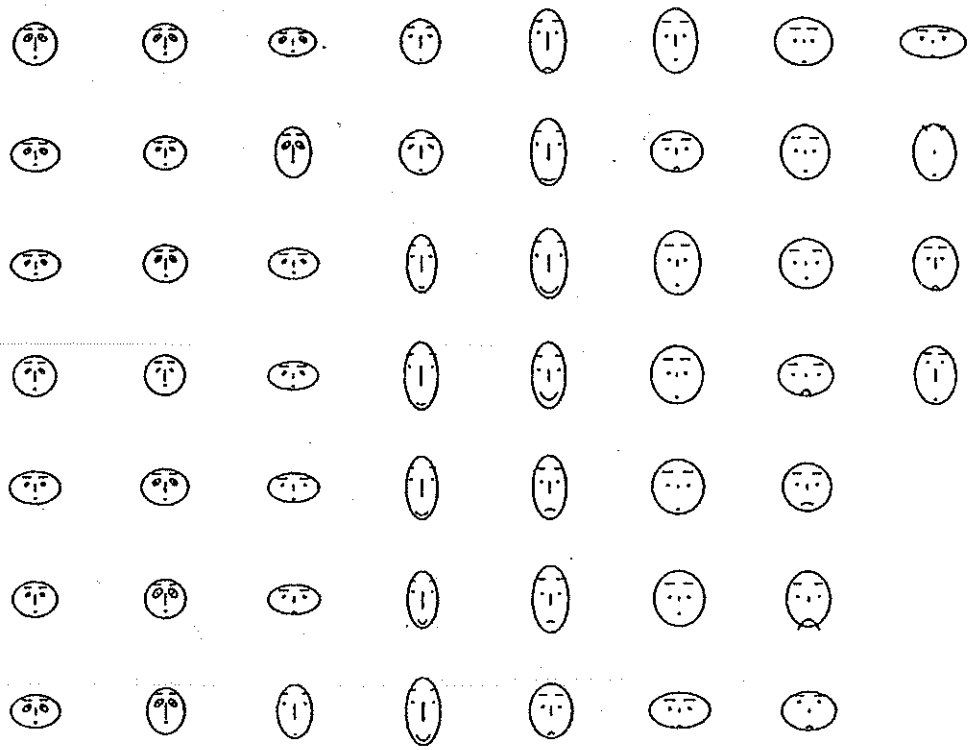


Figure 3.2 Example of abstract dimensional reduction, with each face representing a single twelve-dimensional tuple.
(by Chernoff, reproduced from Crawford & Fall, 1990)

techniques tend to discretise and segregate the individual elements of multidimensional datasets or tuples. Thus, they offer no immediate relevance to the present study.

Combined approaches. Some studies have investigated combining more than a one-dimensional reduction method. McDonald (1983, p. 181), for example, notes that three-dimensional scatterplots can display more dimensions by replacing points at a three-dimensional position with glyphs, each "with colour, size and shape."

Integration of analytic tools is also considered by Beshers and Feiner (1993) who developed *AutoVisual*, a visualisation system reliant upon perspective projection of three-dimensional graphs, together with graphical tools for specifying values of other dimensions in the dataset.

Similarly, Pratico, Hanson, Xu, Jarvis and Vetter (1992, p. 258) used three-dimensional perspective graphs to display subsets of a dataset. Parametric control over the specification and display of dimensions is

provided by a "world within a world," graphic interface which "permits visualization of a 3D solution surface in an inner world, which can be changed along with a corresponding change of the parameters of a 3D outer world."

Problems with high-dimensional projections. As mentioned above, repeated hyperplanar projection is developed in this study and, since it was introduced to graphics developers by Noll (1967), it is worth noting his four-dimensional perspective/parallel visualisations summary reports that "no profound 'feeling' or insight into the fourth spatial dimension was obtained" (Noll, 1967, p. 469).

During the development of this project, it was anticipated that projections of high-dimensional data and/or objects may be difficult to understand. Certainly, it seems logical that if a two-dimensional image of three-dimensional space can be difficult to understand, then a two-dimensional projection of a four- or higher-dimensional hyperspace will surely be more difficult to understand.

However, the relationship between dimensionality and perceptual complexity need not be proportional. Cole (1993, p. 54) notes:

> Take a standard two-dimensional relationship—say, a graph relating interest to consumer spending. Neither has anything to do with geometry, but you can get a better grasp of the situation by looking at the shape of the line. You can easily see where it peaks or bottoms out. You can see the slope of the curve. The same holds true in five- or even ten-dimensional models. "Logically, it may seem like the geometry is lost, that it's just numbers. ... But the geometry can tell you things that the numbers alone can't: how a curve reaches a maximum, how you can get from there to here." You can see hills, valleys, sharp turns and smooth transitions; holes in a doughnut-shaped nine-dimensional model might indicate realms where no solutions lie.

The present study asserts that, just as three-dimensional visualisation requires the support of visual aids and interface tools for interpretation of visualisations, high-dimensional visualisations should be supported by appropriately designed visual aids and tools which assist the process of analysis and interpretation. If, as Cole (1993, p. 56) notes, "our difficulty

with perceiving higher dimensions is primarily psychological," then a computer visualisation tool, using appropriate facilities, may well be sufficient to bridge the perceived gap between three-dimensional visualisation and that for higher-dimensions.

Even after taking into account Noll's comments about negative experiences, visualisations of high-dimensional space, when appropriately implemented, will still allow useful information to be read from the display.

3.3 Other Literature of Significance to the Study

Interaction is an essential aid to understanding and enhancing computer visualisations (Crawford & Fall, 1990; van de Grind, 1986). As Crawford and Fall (1990, p. 95) point out:

> A sophisticated software environment for the interactive exploration of high-dimensional scatterplots can reveal tremendous insights into the structure of a data set, and the advent of low-cost, high-powered graphics workstations provides today's data analyst with the potential for building such environments.

"Interaction," such as dragging a mouse to simulate rotation of a volume on the screen, is a powerful way to communicate an illusion of three dimensions. Structure and relationships among data being displayed may become apparent during animation/motion of the scene (McDonald, 1983, p. 181).

In order for real-time animation to be realistic, the computing environment chosen for the implementation of the algorithms developed in the present study must be of sufficient power: that is, the environment must support rapid calculation and generation of images. Further, three-dimensional computer graphics facilities, such as perspective and other depth cues should preferably be supported.

The *OpenGL* graphics library, as discussed by Neider, Davis and Woo (1993) and Segal and Akeley (1994), together with the X-Windows system has been chosen as the development environment for the following reasons:

- *OpenGL* is a standard, is windowing system-independent, and is presently supported on a wide range of platforms[4] (Segal & Akeley, 1994);

- X-Windows is also widely available and, depending on the hardware environment, supports *OpenGL*;
- *OpenGL* provides "simple, direct control over the fundamental operations of 3D and 2D graphics" (Segal & Akeley, 1994, p. 1) — a "low-level" framework suitable for extension to allow rendering of high-dimensional tuples;
- The *OpenGL* framework provides for matrix transformations and a wide range of conventional graphics facilities such as colour specification, depth cued colours, hidden line./surface removal and perspective projection—all amounting to a three-dimensional "virtual machine," which may require the visualisation system developed to project to only three, rather than two dimensions; and,
- *OpenGL* is performance oriented: features may be turned on or off depending on the needs and limitations of the program and/or hardware (Segal & Akeley, p. 2).

3.4 Summary

High-dimensional methods and geometry may be used as a solution, or part of a solution, to wide-ranging problems in many fields including statistics, mathematics, engineering and physics. Visualisation of high-dimensional space and data may be invaluable, assisting understanding of high-dimensional datasets in one or more of those fields. High-dimensional visualisation is achievable by one or more of various methods, each with certain applications, drawbacks and advantages. One such method is repeated hyperplanar projection, which may be particularly useful because it may be integrated with existing computer visualisation techniques to yield arbitrary views of a high-dimensional dataset. Sophisticated depth cues and interface facilities may also be incorporated, and would aid an observer's interpretation and understanding of projected images.

An implementation of the high-dimensional system which uses a

---

4. Among these supported systems are IBM's AIX R6000, presently supported at Edith Cowan University, and Silicon Graphics' range of workstations, on which future development may take place.

combination of *OpenGL* and the *X-Windows* windowing system, combines ready-made computer graphics functions with graphical user-interface facilities and widespread portability.

4. Design of the Implementation

Arbitrary views of high-dimensional space and data may be achieved by a generalisation of existing three-dimensional computer graphics processes. In particular, a high-dimensional viewing pipeline may be built by generalising three-dimensional planar projection to n-dimensional hyperplanar projection, and three-dimensional transformations to n dimensions. Some background explanation in these areas, including three-dimensional computer graphics and the mathematics of planar projection, arbitrary views and transformations, are tendered as a foundation for subsequent discussion on the generalised n-dimensional form.

4.1 Overview of Relevant Computer Graphics Processes

Computer graphics techniques for viewing three-dimensional scenes are well established (Foley et al., 1991; Hearn & Baker, 1994; Hill, 1990). Essentially, each technique employs a three-fold process:

- transform the objects to be viewed such that projection is mathematically convenient;
- use planar projection to project the objects to a view-plane; and,
- map the projection of the object to the output device.

Although this viewing process is simplified — all notion of "clipping" and hidden surface removal, for example, have been omitted (see "Assumptions and Limitations of the Study" on page 11) — these three steps are all that is required to render an image of an arbitrary view of a three-dimensional object or dataset (Foley et al., 1991).

How then, for example, should an object be transformed to be convenient for projection, and how are arbitrary views of an object specified? The answers to these questions are, respectively, transformations and viewing parameters, both of which are discussed briefly in the following sections, after the introduction of some terminology.

4.1.1 Terminology

Throughout this chapter, the terminology may differ slightly from that of existing three-dimensional computer graphics literature because it is generalised with respect to $n > 3$ dimensions. To illustrate this situation, the following terminology is therefore introduced:

- a *world space* is a n-dimensional "world" or space, within which an object or dataset is defined;
- a *projective space* is a $(n-1)$-dimensional subspace of the world space, onto which the object or dataset may be projected.

The name "projective space" differs from the terminology of other literature: for example, Foley et al. (1991) refer to the "view-plane." In the author's opinion the term "plane" is inappropriate for use in this study because it specifically has connotations for a two-dimensional space or plane. Furthermore, the term "view" is considered inappropriate because repeated projection is used in this study, so any single projective space may not be for "viewing" but an intermediate step toward viewing.

4.1.2 Transformations

Transformations in computer graphics are important because they assist operations such as viewing, interaction and animation, as well improving performance (Neider, Davis & Woo, 1993, p. 109). A two-dimensional point $\mathbf{v} = (x, y)^{\mathrm{T}}$, may be moved, rotated, scaled or otherwise changed by applying a mathematical function to that point (Foley et al., 1991, p. 201; Hill, 1990). The point $\mathbf{v}$ may be moved, for example, along one or more axes by adding or subtracting from the components of the point, yielding the transformed point $\mathbf{v}'$.

An example of a transformation is movement, or *translation*, of a point in space, which may be expressed using vector addition such as

$$\mathbf{v}' = \mathbf{v} + \mathbf{t} = \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \end{bmatrix} = \begin{bmatrix} x + t_x \\ y + t_y \end{bmatrix}$$

where $\mathbf{t}$ is the translation vector, indicating the amount of movement along each axis.

Other "primitive" transformations include *scaling*, *rotation* and *shearing*, as shown in Figure 4.1, each of which may be expressed in matrix form. For example, the matrix to scale by s_x along the x-axis, and s_y along the y-axis is

$$\mathbf{S}(s_x, s_x) = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix};$$

the matrix to rotate about the origin by θ is

$$\mathbf{R}(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix};$$

and, finally, the matrix to shear by sh_x along the x-axis, and by sh_y along the y-axis is

$$\mathbf{SH}(sh_x, sh_y) = \begin{bmatrix} 1 & sh_x \\ sh_y & 1 \end{bmatrix}.$$

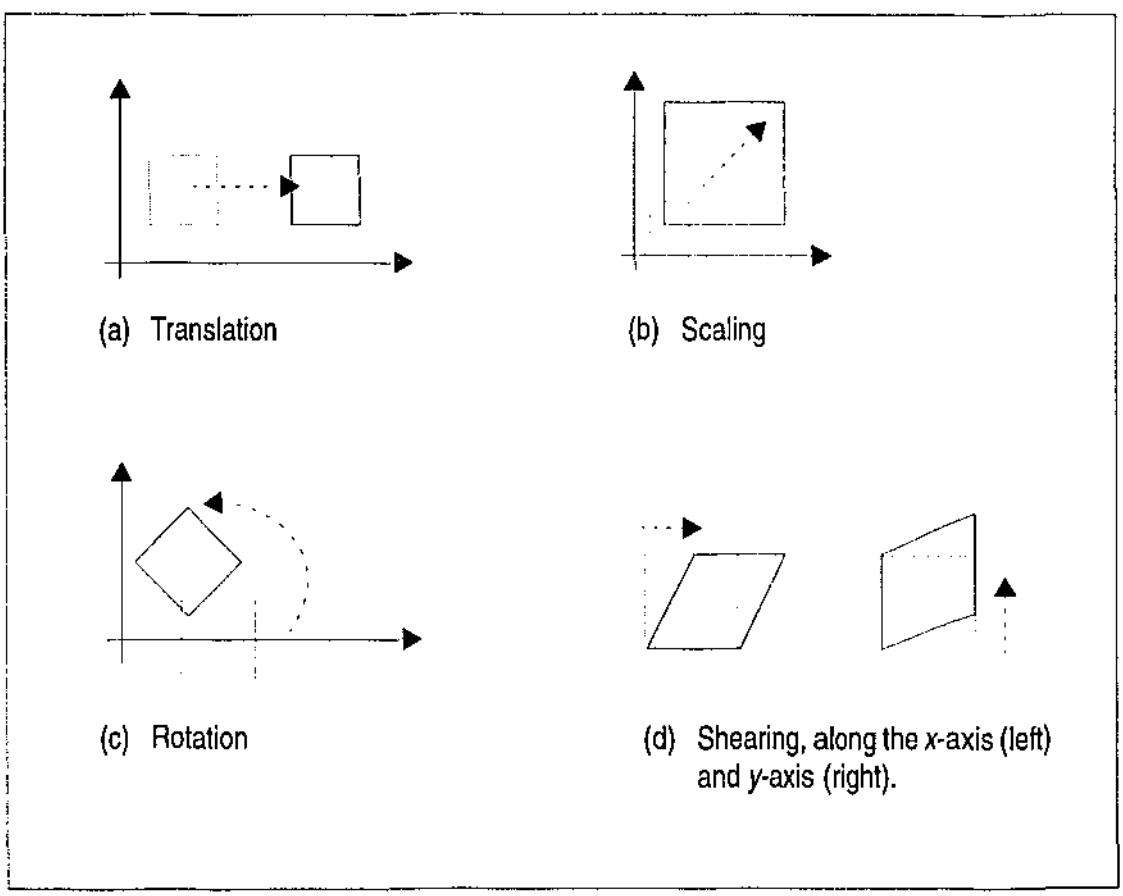


Figure 4.1 Primitive Transformations Applied to a Square.

Further background on the derivation of these matrices may be found in Foley et al. (1991, pp. 201–226).

If **M** is a transformation matrix, then the point **v** may be transformed by post-multiplying[1] **M** by **v** to yield the transformed point **v'**:

$$\mathbf{v}' = \mathbf{M} \cdot \mathbf{v}.$$

Similarly, a two-dimensional object **H**, represented by m vertices $\mathbf{H} = (\mathbf{v}_1, \mathbf{v}_2, ..., \mathbf{v}_m)$ where $\mathbf{v}_i = (x_i, y_i)^T$ for all i in $1..m$, may be moved, rotated, scaled or otherwise transformed as a whole, by multiplying each vertex in the object by a transformation matrix.

To illustrate transformation using vectors and matrices, consider scaling the distance from the origin to the point $\mathbf{v} = (1.5, 1.0)^T$ by 200%. This may be expressed as

$$\mathbf{v}' = \begin{bmatrix} 2.0 & 0.0 \\ 0.0 & 2.0 \end{bmatrix} \cdot \begin{bmatrix} 1.5 \\ 1.0 \end{bmatrix},$$

yielding the result $\mathbf{v}' = (3, 2)^T$.

Successive transformations may be expressed by multiplying, or composing, matrices representing primitive transformations into a single composite matrix. For example, a rotation by 45°, followed by 150% increase in size, may be represented by a single matrix **M**, where

$$\mathbf{M} = \mathbf{S}(1.5, 1.5) \cdot \mathbf{R}(45°) = \begin{bmatrix} 1.5 & 0 \\ 0 & 1.5 \end{bmatrix} \cdot \begin{bmatrix} \cos 45° & -\sin 45° \\ \sin 45° & \cos 45° \end{bmatrix},$$

yielding the result

$$\mathbf{M} = \begin{bmatrix} 1.0605 & -1.0605 \\ 1.0605 & 1.0605 \end{bmatrix}.$$

A point, multiplied by the matrix defined above, would first be rotated by 45°, then scaled about the origin by a factor of 1.5.

Note, however, that a translation cannot be expressed in matrix form, and hence cannot be composed with other transformations. To solve this problem, *homogeneous coordinates* may be used. In homogeneous form, a

---

1. Post-multiplication is used by *OpenGL* (Neider et al., 1993, p. 78) and by Foley et al. (1991, p. 205), but Foley notes that other texts use pre-multiplication with the matrices transposed.

translation may be expressed as a matrix, and applied to a point in space by matrix multiplication as before (Foley et al., 1991, p. 204; Coxeter, 1969).

Homogeneity requires the incorporation of an extra dimension into the arithmetic. Thus, a two-dimensional vector $\mathbf{v} = (x, y)^{\mathrm{T}}$ becomes the two-dimensional homogeneous point $\mathbf{v}^{\mathrm{h}} = (x, y, w)^{\mathrm{T}}$, where w is termed the *homogeneous component*. Initially, w is given the value one. That is, the two-dimensional point

$$\mathbf{v} = \begin{bmatrix} x \\ y \end{bmatrix}$$

is equivalent to the homogeneous point

$$\mathbf{v}^{\mathrm{h}} = \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}.$$

Some transformations, such as projection, give w a value other than one. A homogeneous vector is converted back to two-dimensions by firstly dividing the vector by w, and then dropping the homogeneous component from the definition (Foley et al., 1991, p. 204). That is, the homogeneous point

$$\mathbf{v}^{\mathrm{h}} = \begin{bmatrix} x \\ y \\ w \end{bmatrix}$$

is equivalent to the two-dimensional point

$$\mathbf{v} = \begin{bmatrix} x/w \\ y/w \end{bmatrix}.$$

A similar process is used to homogenise two-dimensional matrices: an extra row and column are added, resulting in matrices with three rows and columns. Two-dimensional translation and scaling are expressed, respectively, by the homogeneous matrices

$$\mathbf{T}(t_x, t_y) = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix}, \qquad \mathbf{S}(s_x, s_x) = \begin{bmatrix} s_x & 0 & 0 \\ o & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix},$$

while rotation and shearing are expressed by the matrices

$$\mathbf{R}(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}, \text{ and } \quad \mathbf{SH}(sh_x, sh_x) = \begin{bmatrix} 1 & sh_x & 0 \\ sh_y & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

Further background to homogeneous coordinates may be found in Foley et al. (1991).

4.1.3 View parameters

View parameters enable arbitrary views of an object or dataset to be calculated and rendered. The view parameters used in this study are[2]:

- the view-point $\mathbf{vp}^{(n)}$ is a point in n-dimensional space whence the object is to be viewed. That is, $\mathbf{vp}^{(n)}$ is where the "observer" or "synthetic camera" is located (Foley et al., 1991; Hill, 1990);
- the target-point $\mathbf{tp}^{(n)}$ is a point in n-dimensional space which is the "centre of projection," or where the observer is focused; and,
- the orientation of the projective space is given by orthonormal basis matrix $\mathbf{PSN}^{(n)} = (\mathbf{psn}_1, \mathbf{psn}_2, \ldots, \mathbf{psn}_n)$. Each vector $\mathbf{psn}_i$ denotes the ith projective space normal and is a n-dimensional vector $(1 \le i \le n)$. Another way to consider each $\mathbf{psn}_i$ is as an axis local to the projective space (Goldman, 1992; Sommerville, 1958).

In order to understand how view parameters enable arbitrary views, recall that three-dimensional planar projection is mathematically convenient for a *normalised view*,[3] so it may be defined by the following constraints:

---

2. Viewing parameter names, such as "projective space" and **psn** differ from other literature, such as Foley et al. (1991), where the equivalent term in VPN (view-plane normal). See *Terminology* on page 28.

3. Foley et al. (1991, p. 259) discuss the *canonical view volume*, equivalent in concept to the normalised view mentioned here. The definitions of the two differ because the canonical view volume imposes size, as well as position and orientation constraints. A size constraint, while useful, is beyond the requirements of the present study (see "A problem with repeated projection" on page 44).

- $\mathbf{vp}^{(n)}$ is defined at r along the nth axis: e.g. $\mathbf{vp}^{(n)} = (0, 0, r)^{\mathrm{T}}$, when $n = 3$;
- $\mathbf{tp}^{(n)}$ is defined along the nth axis: e.g. $\mathbf{tp}^{(n)} = (0, 0, f)^{\mathrm{T}}$, when $n = 3$;
- the projective-space is aligned with the principal axes, and is, in particular, perpendicular to the nth axis of a n-dimensional world space: e.g. $\mathbf{PSN}^{(3)}$ is the three-dimensional identity matrix.

When any or all of the parameters do not adhere to these constraints, transformations may be employed to position and orient the coordinate system such that the arbitrary view becomes the normalised view. Thus, the viewing parameters effectively parameterise the viewing process by defining transformation matrices that map an arbitrary view so that it is ready for projection to the view-plane. The form of the parameterisation and matrix construction is the purpose of the discussion in Section 4.3.

4.1.4 Summary of Relevant Computer Graphics Processes

In summary, then, arbitrary views of a three-dimensional object may be accomplished by using viewing parameters to construct transformation matrices which are applied to every vertex of an object or every datum of a dataset. The matrices are defined in homogeneous form so that any number of rotations, translations and other transformations, as well as projection to the view plane, may be combined into a single matrix. The two-dimensional projection of the object may then be mapped to the output device. A "conceptual model" of this process is shown in Figure 4.2.

4.2 Projection

Discussion of three- and higher-dimensional projections continues after seminal consideration of projection of a two-dimensional world space onto a one-dimensional projective space.

4.2.1 Two-Dimensional Projection to One-Dimension

Although at first it may appear counter-productive to consider projection to a one-dimensional space — a line — an understanding of this relatively straightforward case may aid understanding of subsequent discussion of higher-dimensional projection.

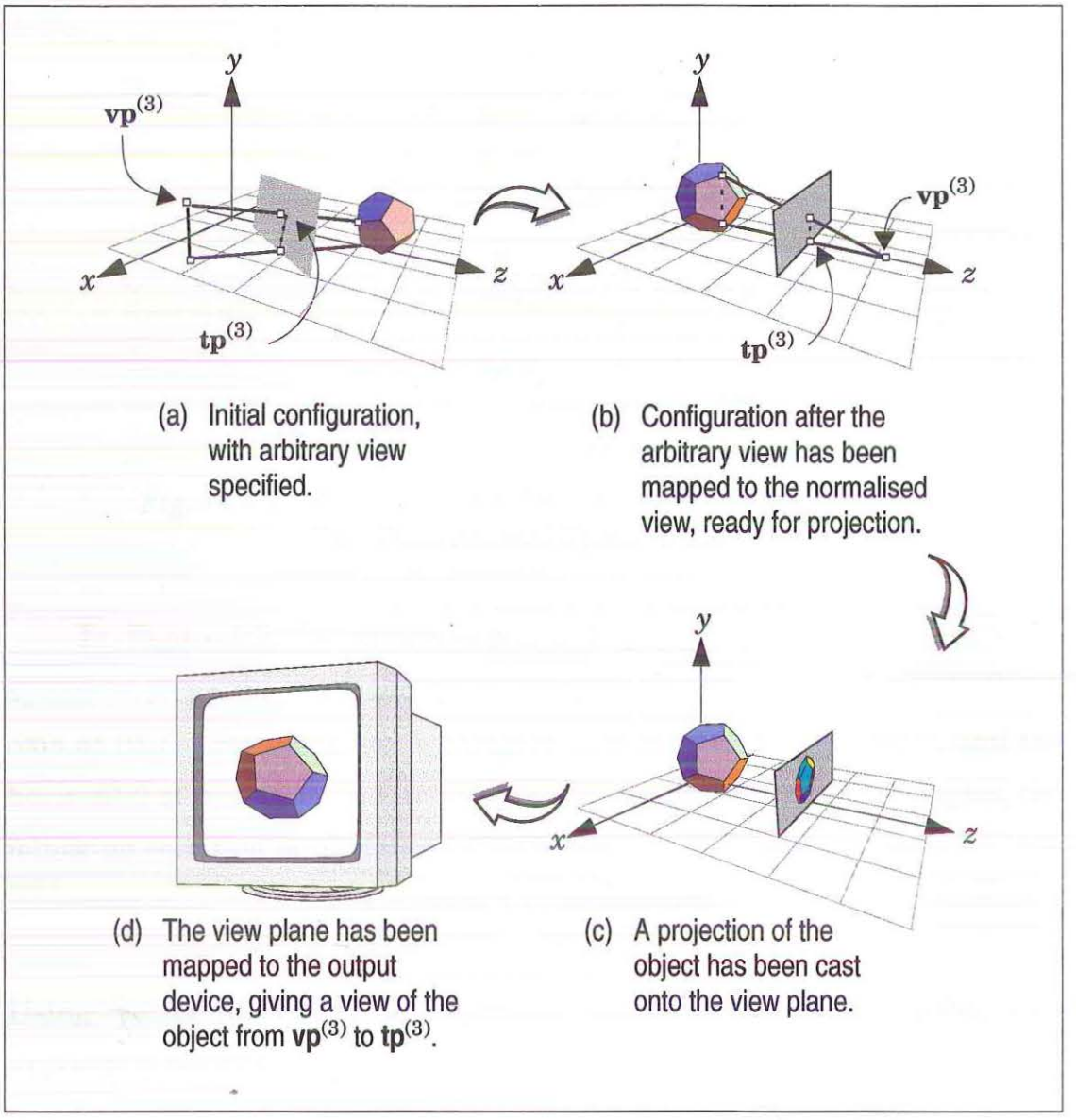


Figure 4.2 Overview of the Three-Dimensional Viewing Process.

Projection of a two-dimensional world space may be considered as follows: within the world space, defined by axes x_1 and x_2, a one-dimensional projective space (a line) is placed at the target-point $\mathbf{tp}^{(2)} = (0, f)^T$ between the observer view-point $\mathbf{vp}^{(2)} = (0, r)^T$ and any points in the space to be projected. Note that the projective space is perpendicular to the x_2 axis, as shown in Figure 4.3. An imaginary straight line, or *projector* (Foley et al., 1991, p. 230), is calculated from every point $\mathbf{p} = (x, y)^T$ to $\mathbf{vp}^{(2)}$. The projector intersects the projective space at $\mathbf{p}'$, which is termed the *projection* of $\mathbf{p}$. The set of all projections represents the projection of the entire dataset.

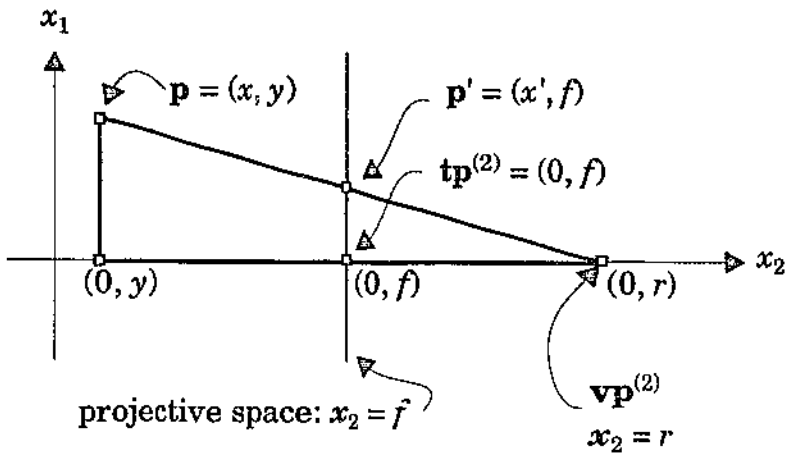


Figure 4.3 Projection of a Two-Dimensional Point onto a One-Dimensional Space (a Line).

In, Figure 4.3, the vertical of **p** is from **p** to the projection of **p** onto the x_2 axis at $(0, y)$, the vertical of **p'** is from **p'** to the projection of **p'** onto the x_2 axis at $(0, f)$. Similarly, the horizontal of **p** is from $(0, y)$ to $\mathbf{vp}^{(2)}$, and the horizontal of **p'** is from (0, f) to $\mathbf{vp}^{(2)}$. Thus, using similarity of triangles, the following equation for distance ratios holds:

$$\frac{\text{vertical of } \mathbf{p}'}{\text{horizontal of } \mathbf{p}'} = \frac{\text{vertical of } \mathbf{p}}{\text{horizontal of } \mathbf{p}}.$$

Using vector modulus[4] to represent the triangle side lengths, this expression becomes

$$\frac{\left\| \begin{bmatrix} x' \\ f \end{bmatrix} - \begin{bmatrix} 0 \\ f \end{bmatrix} \right\|}{\left\| \begin{bmatrix} 0 \\ r \end{bmatrix} - \begin{bmatrix} 0 \\ f \end{bmatrix} \right\|} = \frac{\left\| \begin{bmatrix} x \\ y \end{bmatrix} - \begin{bmatrix} 0 \\ y \end{bmatrix} \right\|}{\left\| \begin{bmatrix} 0 \\ r \end{bmatrix} - \begin{bmatrix} 0 \\ y \end{bmatrix} \right\|}$$

which is equivalent to

$$\frac{\left\| \begin{bmatrix} x' \\ 0 \end{bmatrix} \right\|}{\left\| \begin{bmatrix} 0 \\ r - f \end{bmatrix} \right\|} = \frac{\left\| \begin{bmatrix} x \\ 0 \end{bmatrix} \right\|}{\left\| \begin{bmatrix} 0 \\ r - y \end{bmatrix} \right\|}.$$

4. The modulus of a vector, denoted $\|\mathbf{v}\|$, represents the length of the vector **v** (Foley et al., 1991, p. 1095).

Therefore,

$$\frac{x'}{r-f} = \frac{x}{r-y}$$

which, rearranging to solve for **x'**, yields the result for the one-dimensional projection of a two-dimensional point:

$$x = \left(\frac{r-f}{r-y}\right)x. \tag{4.1}$$

Note that although Figure 4.3 shows **p** above the x_2 axis, similar triangle ratios — and therefore also equation 4.1 — remain defined if **p** is on, or below, the x_2 axis.

Homogeneous representation of two-dimensional projection. Recall from Section 4.1 that the use of homogeneous matrices may simplify the overall viewing process because successive transformations may be combined into a single matrix (Foley et al., 1991). Two-dimensional projection, as defined above, may be expressed by the homogeneous matrix

$$\mathbf{M}_{\text{proj}}^{(2)} = \begin{bmatrix} r-f & 0 & 0 \\ 0 & -f & rf \\ 0 & -1 & r \end{bmatrix}. \tag{4.2}$$

This matrix may be composed with other transformations if needed, or used to project vectors on its own. The superscript (2) indicates the world-dimension relevant to the matrix, thus $\mathbf{M}_{\text{proj}}^{(2)}$ is a matrix defined to carry out projection from two-dimensions to one-dimension.

The definition of $\mathbf{M}_{\text{proj}}^{(2)}$ is verified by considering the projection of **p** via matrix multiplication: **p** must first be extended to the homogeneous form $\mathbf{p}^{\text{h}}$ by incorporating the homogeneous component $w = 1$,

$$\mathbf{p}^{\text{h}} = \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}.$$

The homogeneous point is then transformed by the multiplication

$$\mathbf{p}'^{\text{h}} = \mathbf{M}_{\text{proj}}^{(2)} \cdot \mathbf{p}^{\text{h}}$$

to yield the projected homogeneous point

$$\mathbf{p}'^{\mathrm{h}} = \begin{bmatrix} x(r-f) \\ f(r-y) \\ r-y \end{bmatrix}.$$

Now recall that a two-dimensional homogeneous vector $\mathbf{p}^{\mathrm{h}} = (x, y, w)^{\mathrm{T}}$ is converted to the two-dimensional vector $\mathbf{v} = (x/w, y/w)^{\mathrm{T}}$. So, dividing $\mathbf{p}'^{\mathrm{h}}$ by $w = (r-y)$ effects the projection:

$$\frac{\mathbf{p}'^{\mathrm{h}}}{r-y} = \begin{bmatrix} \left(\dfrac{r-f}{r-y}\right) x \\ f \\ 1 \end{bmatrix}.$$

The third component of the vector is now redundant and may be dropped (Foley et al., 1991, p. 204), yielding the result for $\mathbf{p}'$

$$\mathbf{p}' = \begin{bmatrix} \left(\dfrac{r-f}{r-y}\right) x \\ f \end{bmatrix}. \tag{4.3}$$

Note that although the above expression for $\mathbf{p}'$ is a two-dimensional vector, the second component, f, is constant. so a set of projected points would form a one-dimensional space—this result concurs with that of equation 4.1.

4.2.2 Three-Dimensional Planar Projection

Planar projection of a three-dimensional object or dataset may be considered as follows: a two-dimensional projective space (i.e. a projection plane), is placed between $\mathbf{vp}^{(3)}$ and the object or dataset to be viewed. The projective-space is situated at $\mathbf{tp}^{(3)}$, and oriented such that it is perpendicular to the z-axis. For every point $\mathbf{p}$ in the three-dimensional world, a projector is cast from that point, through the projective-space, to $\mathbf{vp}^{(3)}$. The projector intersects the projective-space at the point $\mathbf{p}'$, which is termed the *projection* of $\mathbf{p}$ onto the plane. The set of all line segments joining the intersection points represents the projection of the entire object. Figure 4.4 shows the three-dimensional planar projection scenario.

As with two-dimensional projection considered earlier, $\mathbf{p}'$ may be found by considering the similar triangles formed by $\mathbf{p}$, the projection of $\mathbf{p}$ onto the

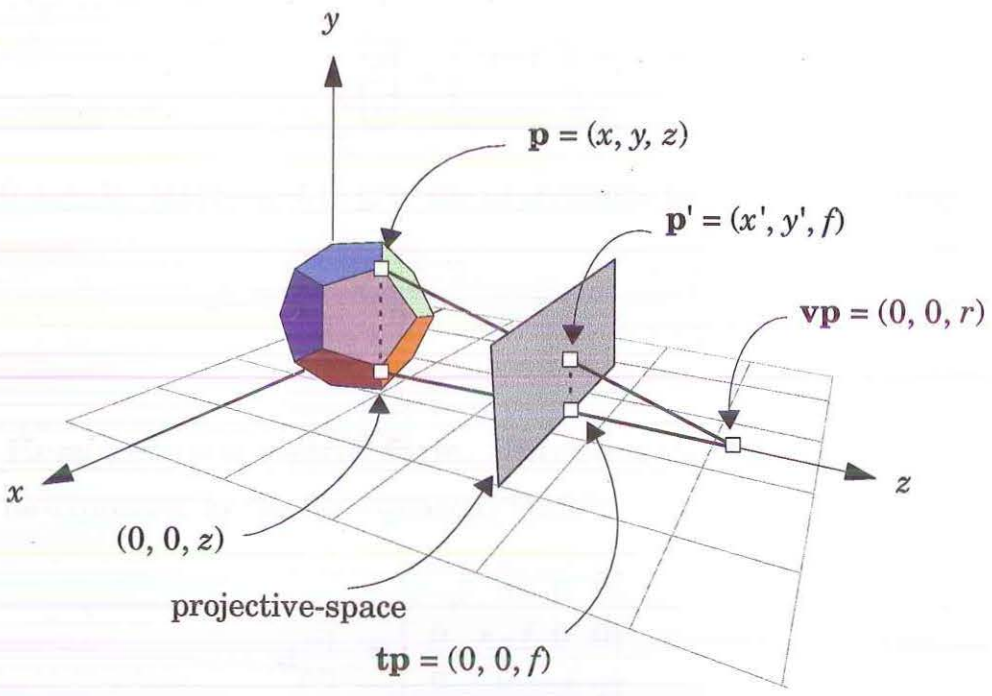


Figure 4.4 Three-Dimensional Planar Projection Scenario.

z-axis and $\mathbf{vp}^{(3)}$, and $\mathbf{p}'$, the projection of $\mathbf{p}'$ onto the z-axis and $\mathbf{vp}^{(3)}$. Although this scenario is three-dimensional, the similarity of these two triangles may still be expressed by the ratios

$$\frac{\text{vertical of } \mathbf{p}'}{\text{horizontal of } \mathbf{p}'} = \frac{\text{vertical of } \mathbf{p}}{\text{horizontal of } \mathbf{p}}.$$

Incorporating vectors, this expression becomes

$$\frac{\left\| \begin{bmatrix} x' \\ y' \\ f \end{bmatrix} - \begin{bmatrix} 0 \\ 0 \\ f \end{bmatrix} \right\|}{\left\| \begin{bmatrix} 0 \\ 0 \\ r \end{bmatrix} - \begin{bmatrix} 0 \\ 0 \\ f \end{bmatrix} \right\|} = \frac{\left\| \begin{bmatrix} x \\ y \\ z \end{bmatrix} - \begin{bmatrix} 0 \\ 0 \\ z \end{bmatrix} \right\|}{\left\| \begin{bmatrix} 0 \\ 0 \\ f \end{bmatrix} - \begin{bmatrix} 0 \\ 0 \\ z \end{bmatrix} \right\|}$$

which is equivalent to

$$\frac{\left\| \begin{bmatrix} x' \\ y' \\ 0 \end{bmatrix} \right\|}{\left\| \begin{bmatrix} 0 \\ 0 \\ r-f \end{bmatrix} \right\|} = \frac{\left\| \begin{bmatrix} x \\ y \\ 0 \end{bmatrix} \right\|}{\left\| \begin{bmatrix} 0 \\ 0 \\ r-z \end{bmatrix} \right\|}.$$

This expression simplifies to yield

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \left(\frac{r-f}{r-z} \right) \begin{bmatrix} x \\ y \end{bmatrix},$$

which may be rearranged to give the expressions for x' and y': namely,

$$x' = \left(\frac{r-f}{r-z} \right) x, \qquad y' = \left(\frac{r-f}{r-z} \right) y. \tag{4.4}$$

Homogeneous matrix form. Three-dimensional planar projection may be expressed by the homogeneous matrix

$$\mathbf{M}_{\text{proj}}^{(3)} = \begin{bmatrix} r-f & 0 & 0 & 0 \\ 0 & r-f & 0 & 0 \\ 0 & 0 & -f & rf \\ 0 & 0 & -1 & r \end{bmatrix}. \tag{4.5}$$

If the homogenised point is defined as $\mathbf{p}^h = (\text{x, y, z, 1})^{\text{T}}$, then the projected homogenised point $\mathbf{p}'^h$ is obtained by:

$$\mathbf{p}'^h = \mathbf{M}_{\text{proj}}^{(3)} \cdot \mathbf{p}^h = \begin{bmatrix} x(r-f) \\ y(r-f) \\ f(r-z) \\ r-z \end{bmatrix}.$$

Division by the homogeneous component, in this case $(r-z)$, and then dropping that component to return to three-dimensions, the projection is effected:

$$\mathbf{p}' = \left[x\left(\frac{r-f}{r-z} \right) \quad y\left(\frac{r-f}{r-z} \right) \quad f \right]^{\text{T}} \tag{4.6}$$

This concurs with equation 4.4.

Verification of three-dimensional projection matrix. It should be noted that the homogeneous matrix for three-dimensional perspective projection, as defined by equation 4.5, is a general form of that provided in computer graphics literature, including Foley et al. (1991). Specifically, the projection Foley et al. (1991) consider assumes the view-point is at $z = -d$ from the origin and the view-plane at $z = 0$. That is, $r = -d$ and $f = 0$. When

applied to equation 4.5, these values yield the perspective matrix

$$\mathbf{M}'_{\text{per}} = \begin{bmatrix} -d & 0 & 0 & 0 \\ 0 & -d & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & -d \end{bmatrix}.$$

Dividing each element by $-d$ gives

$$\mathbf{M}'_{\text{per}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1/d & 1 \end{bmatrix}, \tag{4.7}$$

which is the same perspective matrix as given by Foley et al. (1991, p. 255).[5]

Similarly, recall that *parallel projection* is where the view-point is infinite ($r = \infty$); thus, the distance d between the view-point and the projection plane is also infinite. The matrix defined in equation 4.8 carries out three-dimensional parallel projection (where $\lim_{d \to \infty} \left(\frac{1}{d} \right) \to 0$), and concurs with the form as detailed in Foley et al. (1991, p. 256):

$$\mathbf{M}_{\text{par}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.8}$$

Repeated projection. To assist forthcoming discussion, consider the repeated projection of a two-dimensional point—itself a projection of a three-dimensional point—onto a one-dimensional subspace.

Consider the three-dimensional point $\mathbf{p}$, already projected to the two-dimensional point $\mathbf{p}'$. By equation 4.3, the re-projection of $\mathbf{p}' = (x', y')^{\mathrm{T}}$, to the one-dimensional point $\mathbf{p}''$ is

$$\mathbf{p}'' = \begin{bmatrix} \left(\dfrac{r-f}{r-y'} \right) x' \\ f \end{bmatrix}.$$

5. Foley et al. (1991, p. 255) consider two slightly different forms of perspective projection, summarised by the matrices $\mathbf{M}_{\text{per}}$ and $\mathbf{M}'_{\text{per}}$. Only $\mathbf{M}'_{\text{per}}$ is relevant here because of the nature of repeated projection (see Section 4.2.4, "Generalised Projection" on page 46).

Substituting the values given by equation 4.4 for x' and y' yields

$$\mathbf{p}'' = \begin{bmatrix} \left(\dfrac{r-f}{r-\left(\dfrac{r-f}{r-z} \right) y} \right) \left(\dfrac{r-f}{r-z} \right) x \\ f \end{bmatrix} \tag{4.9}$$

as the result for the repeated projection of **p** from three- to two- to one-dimension. The second component, f, is constant and may be ignored, resulting in a single component.

At a glance, the general form of equation 4.9 makes sense — the multiplicand of the first component of the tuple (x) incorporates multiples of both the second component (y) and the third component (z).

4.2.3 Four-Dimensional Hyperplanar Projection

Four-dimensional hyperplanar projection is directly analogous to three-dimensional planar projection, and may be considered as follows:

- a three-dimensional projective space or *hyperplane* is placed between $\mathbf{vp}^{(4)}$ and an object or dataset in the four-dimensional world-space to be projected;
- for every point **p** in the world space, a projector is cast from that point, through the projective-space, to $\mathbf{vp}^{(4)}$. The projector intersects the projective-space at the point **p'**, the projection of **p** onto the hyperplane;
- the set of all line segments connecting projected points forms a three-dimensional projection of the world-space.

This configuration is shown in Figure 4.5.

As with lower-dimensional planar projection, a four-dimensional tuple $\mathbf{p} = (x_1, x_2, x_3, x_4)^T$ may be projected to the point **p'** on a three-dimensional hyperplane by considering the similar triangle ratios

$$\frac{\text{vertical of } \mathbf{p}'}{\text{horizontal of } \mathbf{p}'} = \frac{\text{vertical of } \mathbf{p}}{\text{horizontal of } \mathbf{p}},$$

which may be rewritten as

$$\frac{\left\| \begin{bmatrix} x_1' \\ x_2' \\ x_3' \\ f \end{bmatrix} - \begin{bmatrix} 0 \\ 0 \\ 0 \\ f \end{bmatrix} \right\|}{\left\| \begin{bmatrix} 0 \\ 0 \\ 0 \\ r \end{bmatrix} - \begin{bmatrix} 0 \\ 0 \\ 0 \\ f \end{bmatrix} \right\|} = \frac{\left\| \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} - \begin{bmatrix} 0 \\ 0 \\ 0 \\ x_4 \end{bmatrix} \right\|}{\left\| \begin{bmatrix} 0 \\ 0 \\ 0 \\ r \end{bmatrix} - \begin{bmatrix} 0 \\ 0 \\ 0 \\ x_4 \end{bmatrix} \right\|}.$$

Simplifying, and rearranging, yields the result in vector form:

$$\begin{bmatrix} x_1' \\ x_2' \\ x_3' \end{bmatrix} = \left(\frac{r-f}{r-x_4} \right) \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}.$$

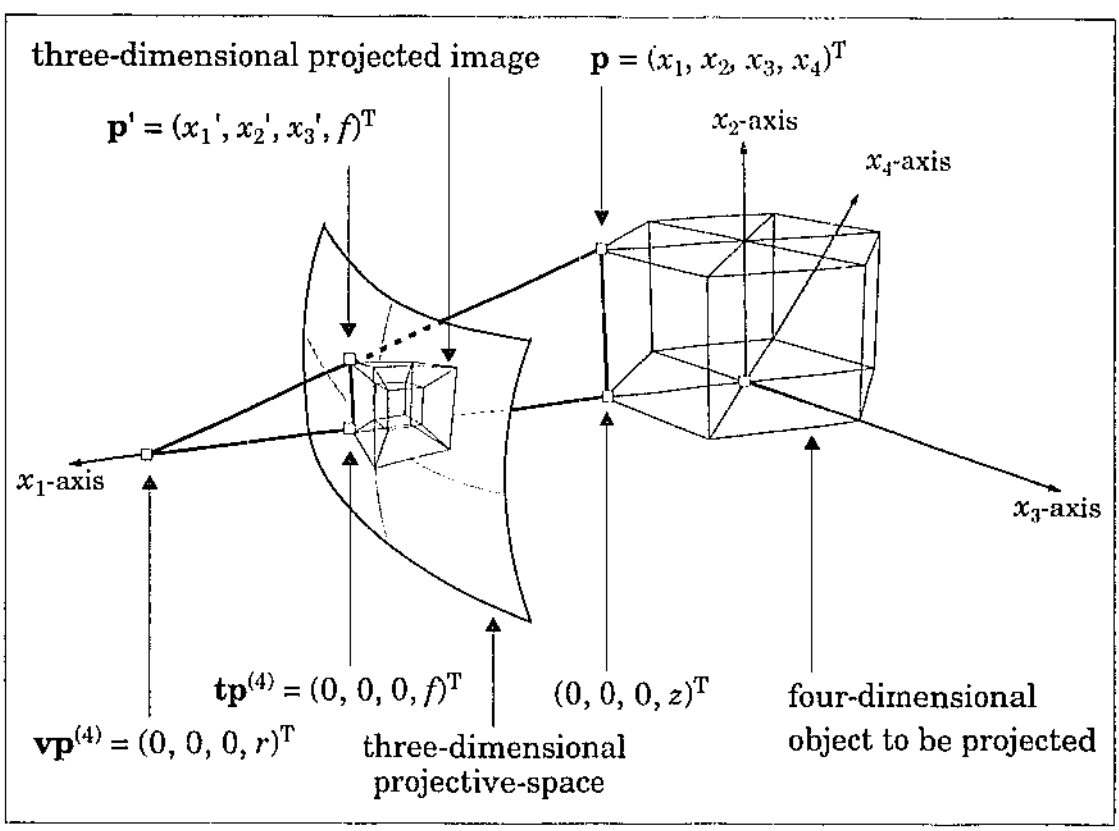


Figure 4.5 Four-Dimensional Perspective Projection.
(Partially based on a figure by Noll, 1967, p. 470)

This, expressed in terms of the components of $\mathbf{p}'$, is

$$\mathbf{p}' = \left[\left(\frac{r-f}{r-x_4} \right) x_1 \quad \left(\frac{r-f}{r-x_4} \right) x_2 \quad \left(\frac{r-f}{r-x_4} \right) x_3 \right]^{\mathrm{T}}. \tag{4.10}$$

Homogeneous representation of four-dimensional projection. Four-dimensional planar projection may be expressed by the homogeneous matrix

$$\mathbf{M}_{\mathrm{proj}}^{(4)} = \begin{bmatrix} r-f & 0 & 0 & 0 & 0 \\ 0 & r-f & 0 & 0 & 0 \\ 0 & 0 & r-f & 0 & 0 \\ 0 & 0 & 0 & -f & rf \\ 0 & 0 & 0 & -1 & r \end{bmatrix}. \tag{4.11}$$

This definition is verified in the same manner as for two- and three-dimensional homogeneous matrices, earlier in this chapter. That is, by multiplying a homogenised general four-dimensional point $\mathbf{p}' = (x_1, x_2, x_3, x_4, 1)^{\mathrm{T}}$ by $\mathbf{M}_{\mathrm{proj}}^{(4)}$, the result concurs with expression 4.10 above.

Repeated projection: Four- to three- to two-dimensions. The previous section establishes that a four-dimensional point $\mathbf{p}$ may be projected to a three-dimensional point $\mathbf{p}'$. Section 4.2.2 establishes that a three-dimensional point may be projected to a two-dimensional point. Combining these two projections—projecting $\mathbf{p}'$ to three-dimensions and then to two-dimensions—yields a two-dimensional point that is straightforward to output on a device.

The two-dimensional projection of a point $\mathbf{p}'$, by equation 4.4, is

$$\mathbf{p}' = \left[\left(\frac{r-f}{r-x_3'} \right) x_1' \quad \left(\frac{r-f}{r-x_3'} \right) x_2' \right]^{\mathrm{T}}.$$

By calculating **p'**, the three-dimensional projection of **p**, using equation 4.10, then substituting the result into equation 4.4, the two-dimensional repeated projection **p"** of a four-dimensional point **p** is

$$\mathbf{p}'' = \left[\frac{(r-f)^2}{\left(r - \frac{r-f}{r-x_4}x_3\right)(r-x_4)}x_1 \quad \frac{(r-f)^2}{\left(r - \frac{r-f}{r-x_4}x_3\right)(r-x_4)}x_2 \right]^{\mathrm{T}}. \tag{4.12}$$

This expression is beginning to look cumbersome, and may be simplified to

$$\mathbf{p}'' = \left[q_2 q_1 x_1 \quad q_2 q_1 x_2 \right]^{\mathrm{T}}$$

by replacing the multiplicands of the first and second projections by q_1 and q_2 respectively. Specifically, these are defined as

$$q_1 = \left(\frac{r-f}{r-x_4}\right), \qquad q_2 = \left(\frac{r-f}{r-x_3'}\right) = \left(\frac{r-f}{r-q_1 x_3}\right) \text{ because } x_3' = \left(\frac{r-f}{r-x_4}\right)x_3$$

Further discussion of repeated projection takes place in Section 4.2.4.

Problems with repeated projection. Two problems with repeated projection should be noted. Firstly, the choice of r and f, (i.e. the locations of $\mathbf{vp}^{(n)}$ and $\mathbf{tp}^{(n)}$ respectively), makes considerable difference to the result of repeated projection. Consider equation 4.12, for example, after defining $r = 0$ and $f = 1$:

$$\mathbf{p}'' = \left[\frac{x_1}{x_3} \quad \frac{x_2}{x_3} \right]^{\mathrm{T}}.$$

This is equivalent to equation 4.10 with $r = 0$ and $f = 1$ — that is, a projection *only* from three-dimensions to two-dimensions. In other words, by placing the view-point at the origin, repeated projection from four- to two-dimensions degenerates into projection from three- to two-dimensions—the fourth component is discarded.

A solution may be to ensure that r is non-zero and f 1. However, Foley et al. (1991, pp. 253 & 279) suggest that the three-dimensional viewing process is mathematically convenient by defining $r = 0$ and $f = 1$. This

approach makes an arbitrary volume in space mathematically straightforward to squeeze into a *canonical volume*[6] and, importantly, to clip portions of the object or dataset that lie outside the volume.

However, because this study is not concerned with clipping operations (see "Assumptions and Limitations of the Study" on page 11), repeated projection is defined only for the inclusive conditions $r = 0$ and $f = 1$.

The second problem with repeated projection is that it is impossible to compose successive projection matrices to effect repeated projection with a single matrix. Stated differently, there is no way to define $\mathbf{M}_{\text{proj}}^{(n-1)'}$ so that it is the same size as $\mathbf{M}_{\text{proj}}^{(n)}$, and the following expression holds:

$$\mathbf{M}_{\text{proj}}^{(n-1)'} \cdot \mathbf{M}_{\text{proj}}^{(n)} \cdot \mathbf{p}^{(n)} = \mathbf{M}_{\text{proj}}^{(n-1)} \cdot \mathbf{p}^{(n-1)}$$

where

$$\mathbf{p}^{(n-1)} = \mathbf{M}_{\text{proj}}^{(n)} \cdot \mathbf{p}^{(n)}.$$

The incompatibility is due to the denominator of each component of a repeat-projected point itself involving division of other high-dimensional components. For example, the multiplicand of each component in expression 4.12 requires the division by a multiple of x_3. The multiple of x_3 itself requires a division by x_4. Such nesting cannot be represented by a single matrix multiplication, and may only be achieved by isolated homogeneous matrix operations: each point is multiplied by the homogeneous matrix, then returned from homogeneous space by dividing the homogenous component (and effecting the projection), *then* multiplied by the next matrix and again divided by the homogeneous component (to effect the repeated projection).

Non-composibility is not of serious consequence because repeated projection may still take place. However, it was hoped that by composing projection matrices the computational efficiency of repeated projection would be improved.

---

6. A canonical volume, meaning standardised or normalised volume, is useful because "certain view volumes are easier to clip against than [a] general one" (Foley et al., 1991, p. 259). That is, if the bounds of the volume are known and, preferably constant, then clipping is mathematically straightforward.

4.2.4 Generalised Projection

Previous discussion has detailed how projection from two-, three- and four-dimensions may be derived by considering similar triangles. The projection may be generalised, allowing a n-dimensional space to be projected to a $(n-1)$-dimensional subspace.

The projection of a n-dimensional point $\mathbf{p} = (x_1, x_2, \ldots, x_n)^T$ onto a projective-space at $x_n = f$, with a viewpoint at $x_n = r$, is given by

$$\mathbf{p}' = \begin{bmatrix} x_1' \\ x_2' \\ \cdots \\ x_{n-1}' \end{bmatrix} = \left(\frac{r-f}{r-x_n}\right) \begin{bmatrix} x_1 \\ x_2 \\ \cdots \\ x_{n-1} \end{bmatrix}. \tag{4.13}$$

This result is also stated by Noll (1967, p. 470). Alternatively, the same projection is achieved by multiplication with the matrix

$$\mathbf{M}_{\text{proj}}^{(n)} = \begin{bmatrix} r-f & 0 & \ldots & 0 & 0 \\ 0 & r-f & \ldots & 0 & 0 \\ \ldots & \ldots & \ldots & \ldots & \ldots \\ 0 & 0 & \ldots & -f & rf \\ 0 & 0 & \ldots & -1 & r \end{bmatrix}, \tag{4.14}$$

where the r component in the lower-left of the matrix is at the $(n+1)$ row and column.

Algorithm development. Points may be projected by multiplication, such as equation 4.13, or by transformation using homogeneous matrices, such as equation 4.14. Algorithms, written in an Ada-like pseudo-code, to project a point by multiplication and transformation are respectively presented in algorithms 4.15 and 4.16.

```
-- Assume definition of "Real_Vector" data structure

function Project_ND_Point_To_N_Minus_1_D
     (Integer n, Real r, Real f, Real_Vector p)
     return Real_Vector is

     Real q := (r-f)/(r-p(n));

begin
     -- result: multiply all but last component by q
     return q * p(1..n-1);
end;
```

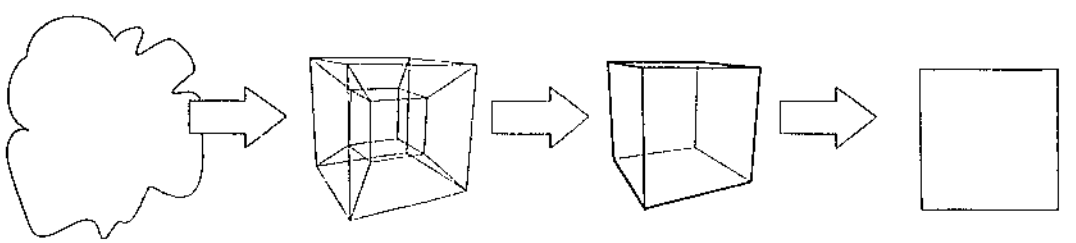
(4.15)

```
-- Assume definition of "Real_Matrix" data structure

function Transform_Vector_By_Homogeneous_Matrix
     (Integer n, Real_Matrix M, Real_Vector p)
return Real_Vector is

     Real_Vector p^h, p^h';
begin
     -- homogenise p
     p^h = new Real_Vector'(1..n => p(1..n), n+1 => 1.0);

     -- transform homogeneous point
     p^h' = M * p^h;

     -- unhomogenise and return

     return  p^h'                (1..n-1);
             ------
             p^h'(n+1)
```

(4.16)

Repeated projection: *n*-dimensions to *k*-dimensions. Recall from chapter two, that Noll (1967, p. 469) notes hyperplanar projection "could be applied repetitively until finally a three-dimensional object representing the successive projections of an n-dimensional hyperobject is obtained." Thus, hyperplanar projection, as defined above, may be used to project a n-dimensional space to a $(n-1)$-dimensional subspace, which may then be repetitively projected $(n-k)$ times, resulting in a k-dimensional hyperplane $(2 \leq k < n)$ suitable for output to a device.

Figure 4.6 illustrates how repeated hyperplanar projection conceptually reduces a n-dimensional space to a lower dimensional subspace. The n-dimensional space is represented as an amorphous shape, indicating the spatial incomprehensibility of such a space.



***n*-dimensional space** may be repeatedly projected to a (*n*-1)-dimensional space

Four-dimensional space may be projected to a three-dimensional space

Three-dimensional space may be projected to a plane

The **Two-dimensional space** of a plane is straightforward to map to a device

Figure 4.6 Conceptual Approach to Projection from High-Dimensions to Lower-Dimensions.

An algorithm to achieve repeated projection might make use of earlier algorithms that project one dimension only, such as algorithms 4.15 and 4.16. Beginning by projecting the n-dimensional vector, the result of each projection may be used as a parameter to the next. Recursive and iterative algorithms for repeated projection, from n- to k-dimensions, are shown in algorithms 4.17 and 4.18 respectively.[7]

```
function Project_ND_Point_To_KD
(     Integer n, Integer k,
      Real r, Real f,
      Real_Vector p
)     return Real_Vector is

begin
      if n = k then -- finished projection
            return p;

      elsif n > k

            return Project_ND_Point_To_KD(
                  n-1, k, r, f,
                  Project_ND_Point_To_N_Minus_1_D(n, r, f, p)
            );

      else -- cannot have k > n
            return NULL_VECTOR; -- as an error signal
      end if;
end;
```

(4.17)

Subspace projections. A n-dimensional point does not have to be

```
function Project_ND_Point_To_KD
(     Integer n, Integer k,
      Real r, Real f,
      Real_Vector p
)     return Real_Vector is

      Real_Vector Result := p;
begin
      while (n > k) do

            Result :=
                  Project_ND_Point_To_N_Minus_1_D (n, r, f, Result)
            n := n - 1;

      end loop;

      return Result;
end;
```

(4.18)

7. These algorithms do not ensure that r is non-zero and that f does not equal one.

projected directly to k-dimensions, where k is the dimensionality of the output device (such as $1 \leq k \leq 3$). Rather, it may be advantageous to project the world-space to a l-dimensional pseudo world-space, where $0 < k < l < n$, and *then* project to the k-dimensional device.

This may be useful as n becomes larger. For example, if $n = 12$, $l = 4$ and $k = 3$, a twelve-dimensional dataset would be projected to four-dimensions and *then* projected to the three-dimensional device. The view of the world-space may be interacted with and visualised. The advantage and drawback of this approach is that two levels of viewing parameters must be defined: one set for the n-dimensional world, and the other for the l-dimensional pseudo-world. Neglecting interaction with the world-space, while simplifying the user's task (because even a four-dimensional view is easier to understand than a higher one), may give a prejudiced view of the dataset. For example, if the n-dimensional dataset is distributed as an array of pencil-like structures in a row, but the view-point is chosen poorly—from the end of a pencil, for example—the low-dimensional projection of the dataset may appear as a line, and give little indication of the greater nature of the dataset.

4.2.5 Summary of Projection

The mathematics of projection, from any dimension, have been established. Several algorithms for achieving repeated projection have also been presented. Projection, however, is still only defined for a constrained view. The mathematical background for incorporating an arbitrary view, prior to projection, is the focus of Section 4.3.

4.3 Arbitrary Views

When an arbitrarily specified volume in space is to be projected, the volume must be rotated and placed such that it conforms to the constraints required for projection.

4.3.1 Arbitrary Views In Two-Dimensions

Although it may initially appear trivial to consider projection of an arbitrary view of a two-dimensional space projected onto a one-dimensional

subspace, such a discussion is a sound starting point for three-, four- and higher-dimensions.

Recall from Section 4.2 that projection is mathematically convenient when:

- the view-point is defined at r along the nth axis of a n-dimensional space: thus, $\mathbf{vp}^{(n)} = (0, 0, ..., r)^{\mathrm{T}}$;
- the target-point is defined along the nth axis of a n-dimensional space: thus, $\mathbf{tp}^{(n)} = (0, 0, ... f)^{\mathrm{T}}$; and,
- the projection-space is defined to be perpendicular to the nth axis of a n-dimensional space. In two-dimensional space, for example, $\mathbf{psn}^{(2)} = (0, 1)^{\mathrm{T}}$.

These constraints, shown graphically in Figure 4.7, limit the view of a dataset or scene considerably. In order to achieve arbitrary views, these constraints must be lifted to allow

- the view-point to be defined anywhere within the n-dimensional space;
- the projection-space to be at any orientation within the n-dimensional space; and,
- the target-point to be defined anywhere within the n-dimensional space.

The removal of each of these constraints is addressed in turn. Note that, throughout this section, **vp** and **tp** refer to $\mathbf{vp}^{(2)}$ and $\mathbf{tp}^{(2)}$ respectively.

1. *Arbitrarily placed view-point.* Consider the viewing configuration shown in Figure 4.8(a)—this scene meets the constraints noted above,

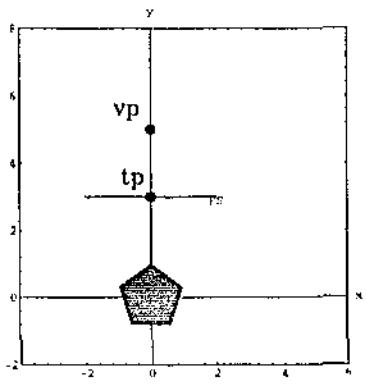


Figure 4.7 Ideal Two-Dimensional Projection Configuration.

except that the view-point is not along the y-axis. A shearing transformation may be used to bring the view-point into alignment.

The shear needs to transform the line from **vp** to **tp** such that it is aligned with the x_2 axis. This line may also be considered as the *direction of projection,* or **dop** (Foley et al., 1991, p. 264). In two-dimensions, **dop** is defined as

$$\mathbf{dop} = \mathbf{vp} - \mathbf{tp} = (vp_x - tp_x, vp_y - tp_y)^{\mathrm{T}}$$

and it must be sheared so that

$$\mathbf{dop}' = (0, vp_y - tp_y)^{\mathrm{T}}.$$

A matrix **S** may be defined to accomplish the transformation, satisfying

$$\mathbf{dop}' = \mathbf{S} \cdot \mathbf{dop}$$

To find the definition of **S**, consider an interim definition of

$$\mathbf{S} = \begin{bmatrix} 1 & s & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

where s is the shearing coefficient and is yet to be defined. Multiplying the homogeneous $\mathbf{dop}^{\mathrm{h}} = (dop_x, dop_y, 1)^{\mathrm{T}}$ by this, and dropping the homogeneous component, would yield

$$\mathbf{dop}' = \begin{bmatrix} dop_x + s \cdot dop_y \\ dop_y \end{bmatrix}.$$

as the result for **dop**'. Because **dop**' must equal $(0, dop_y)$, s must be defined so that $dop_x + s.\ dop_y = 0$. Solving this for s yields $s = -dop_x/dop_y$. The shear matrix **S** is therefore defined as

$$\mathbf{S} = \begin{bmatrix} 1 & -\dfrac{dop_x}{dop_y} & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

Figure 4.8(b) shows the scene after the shear transformation is applied — the scene now fits the projection constraints.

It should be noted that when other view parameters are not within the projection boundaries, as discussed shortly, the view-point, and hence the direction-of-projection, must also be transformed. This scenario is considered in Section 4.3.2, "Arbitrary Views in Three-Dimensions."

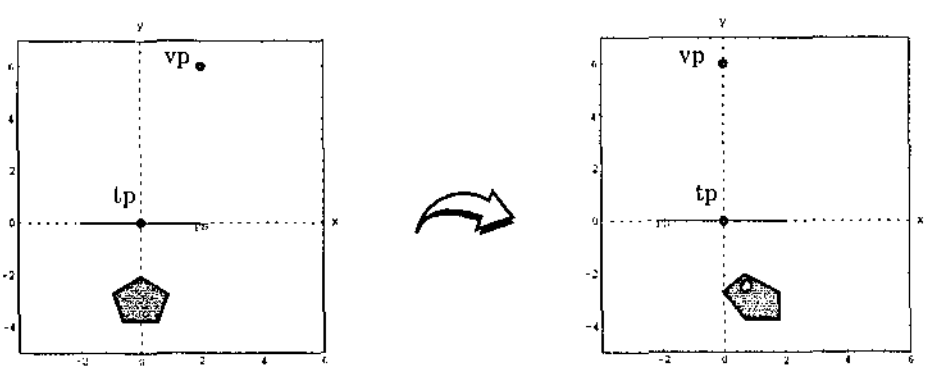
2. *Arbitrarily oriented projective-space.* If the projective-space is not oriented such that it is perpendicular to the y-axis, a rotation may be applied to transform the space to be aligned as required. This configuration is shown in Figure 4.9.

The rotation may be expressed as a matrix **R**, defined as a function of the projective-space normal vectors in **PSN**. The definition of **R** is best considered through reference to both rotation matrices and special orthogonal matrices (Foley et al., 1991, pp. 207 & 220), which have the property that the row vectors of the matrix become aligned with the standard orthogonal basis of the system, and rotation matrices. For the interim, consider the definition of **R** as

$$\mathbf{R} = \begin{bmatrix} a & b & 0 \\ c & d & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

A *special orthogonal matrix* is a matrix where each of the row vectors comprising the matrix is (i) a unit vector and (ii) perpendicular to the other vector(s) (Foley et al., 1991, p. 206). Special orthogonal matrices have various useful properties: the salient property for this application is that the row vectors rotate into the principal axes of a coordinate system when multiplied by the matrix.

That is, multiplying **R** above by $(a, b)^T$ yields $(1, 0)^T$, and multiplying by $(c, d)^T$ yields $(0, 1)^T$. With this knowledge, the values of a, b, c and d can be chosen appropriately.



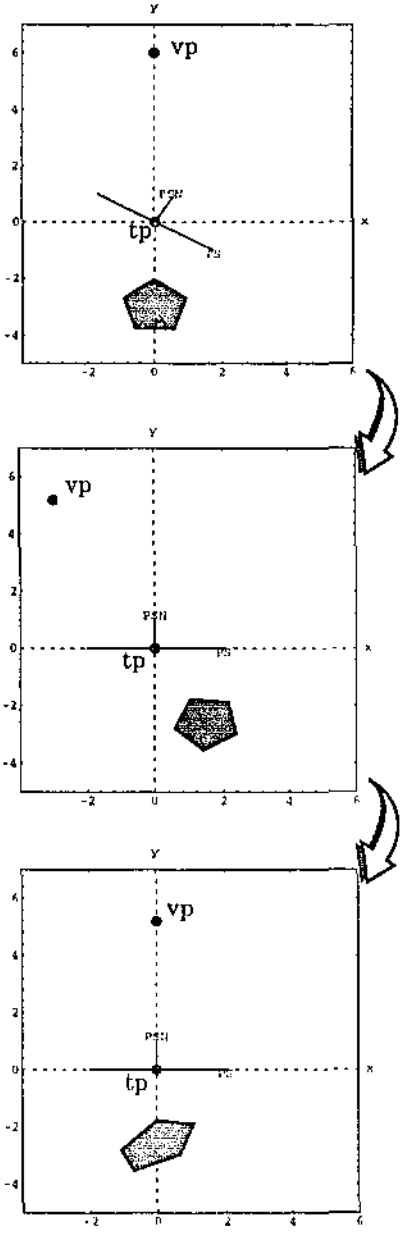
(a) Original viewing configuration, with arbitrarily placed view-point.

(b) After shearing transformation, view-point is aligned with y-axis.

Figure 4.8 Transformation to Align Arbitrary View-Point with y-axis.

The value of the vector to be rotated into the y-axis is considered first: the projection constraints require that the projective-space be oriented normal to the nth axis of the coordinate system. That is, the $\mathbf{psn}_1$ must become $(1, 0)^T$. Therefore, c and d take the values $\mathbf{psn}_{1,x}$ and $\mathbf{psn}_{1,y}$ respectively.

The values of a and b may be determined by referring back to the definition of a two-dimensional rotation. Because c and d now correspond to values of sin t and cos t respectively, the values of a and b, being cos t and -sin t, are straightforward: $a = \mathbf{psn}_{2,y}$ and $b = -\mathbf{psn}_{2,x}$.



(a) Original viewing configuration, with arbitrarily oriented projective space. The orientation is given by the projection space normal (**psn**).

(b) After rotating so that the **psn** is aligned with the *y*-axis. Note that the view-point is now not aligned, as if it had been arbitrarily placed.

(c) After shearing (as before) so that the view-point is aligned with the y-axis, the system is arranged such that projection may be performed.

Figure 4.9 Transformation to Align an Arbitrarily Oriented Projection Space with the *y*-axis.

The rotation matrix is therefore defined

$$\mathbf{R} = \begin{bmatrix} psn_2 & -psn_1 & 0 \\ psn_1 & psn_2 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

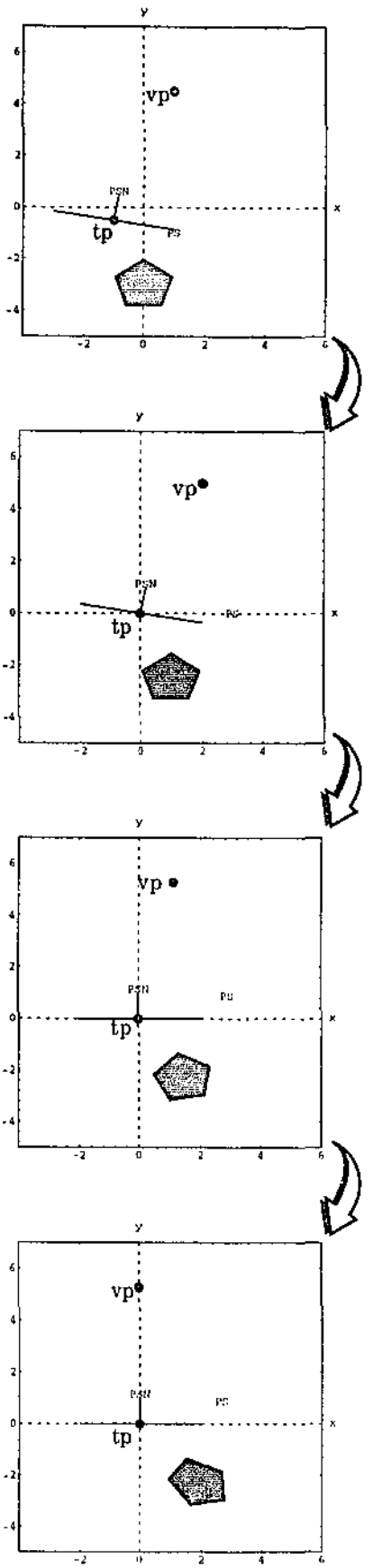
3. *Arbitrarily located target-point.* In order to transform an arbitrarily located target-point so that it conforms to the projection constraints, the system is transformed such that $x = 0$. Further, because rotation and shearing are defined about the origin, the target-point is transformed to be at the origin. The translation matrix to achieve this is

$$\mathbf{T} = \begin{bmatrix} 1 & 0 & -vp_x \\ 0 & 1 & -vp_y \\ 0 & 0 & 1 \end{bmatrix}.$$

Note that the distortion of the object, visible in Figure 4.10(c), reflects the original viewing configuration — the tip of the polygon is still the focus of the scenario, but the right edge (on the page) of the polygon looks as if it has been pulled toward the projective-space. Arbitrarily oriented projective spaces are particularly useful for three-dimensional projection, discussed subsequently.

The two-dimensional arbitrary viewing process. The individual transformation matrices established above may be composed to give a single matrix that transforms an arbitrary view to the projection-ready view, as shown in Figure 4.11. The matrix is defined

$$\mathbf{M}_{\text{Arb}}^{(2)} = \mathbf{S} \cdot \mathbf{R} \cdot \mathbf{T}. \tag{4.19}$$



(a) Original viewing configuration, with arbitrarily placed target-point.

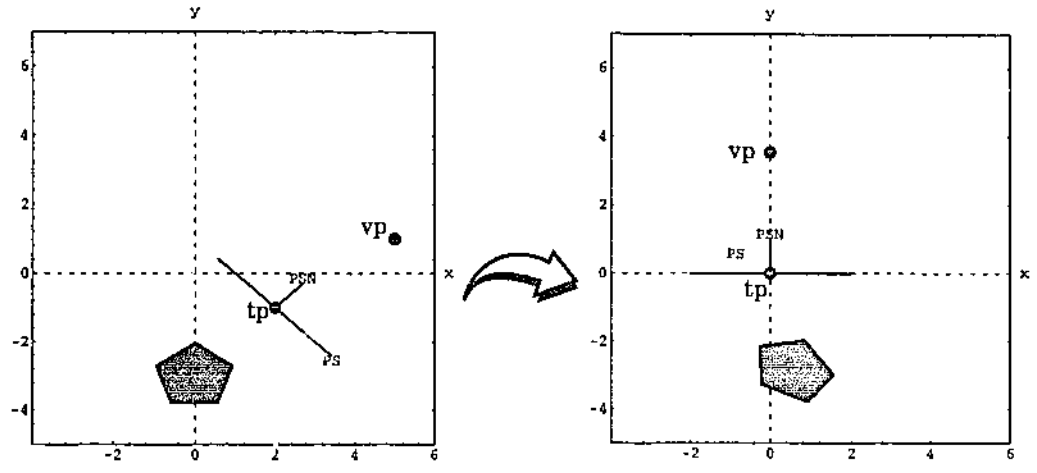
(b) After translating the system so the target point is at the origin rotation and shearing may be applied as before.

(c) Following rotation, the projective-space is now correctly oriented, but the view-point is not aligned correctly with the *y*-axis.

(d) After shearing the view-point is aligned and the system meets the constraints for projection.

The distortion visible of the object reflects the view parameter choice.

Figure 4.10 Transformations to Align Arbitrarily Placed Target-Point.



(a) An arbitrary two-dimensional view.

(b) Following multiplication by $\mathbf{M}_{Arb}^{(2)}$ the system meets projection constraints.

Figure 4.11 The Effect of the Arbitrary Two-Dimensional View Alignment Matrix.

4.3.2 Arbitrary Views in Three-Dimensions

The view parameters, specific for three-dimensional space, are summarised as follows:

- a view-point $\mathbf{vp}^{(3)} = (vp_x,\ vp_y,\ vp_z)^T$ is a point defined in three-dimensional space, and is the point from which the dataset or object will be viewed;
- the target-point $\mathbf{tp}^{(3)} = (tp_x,\ tp_y,\ tp_z)^T$ is a point defined in three-dimensional space, and is the point that will be at the centre of the final image;
- the projective space is a two-dimensional space centred at $\mathbf{tp}^{(3)}$, oriented to be vertically aligned with $\mathbf{psn}_1^{(3)}$ (termed view-up-vector or VUP by Foley et al.), and normal to $\mathbf{psn}_2^{(3)}$ (termed the view-plane normal by Foley et al.).

An arbitrary view within three-dimensional space is a view where these view parameters have not been constrained by the projection requirements, as defined in Section 4.1.3. As with the two-dimensional case, projection may only be carried out after transforming the system so that it conforms to these constraints. The series of transformations

required is directly analogous to that of the two-dimensional process, and may be considered as the following three steps:

- translate the system so the target-point $\mathbf{tp}^{(3)}$ is at the origin. This meets the requirement that f is on the z-axis (in fact $f = 0$), and ensures that subsequent transformations are correctly defined[8];
- rotate the system so that the projective-space is perpendicular to the z-axis, and is vertically oriented. That is, the local axes of the projective space are aligned with the principal axes: $\mathbf{psn}_1^{(3)}$ becomes $(0, 1, 0)^T$ and $\mathbf{psn}_2^{(3)}$ becomes $(0, 0, 1)^T$; and,
- shear the system so that $\mathbf{vp}^{(3)}$ is aligned with the z-axis. A shear, rather than some other rotation or translation, is performed so that the parameters already correctly aligned (namely $\mathbf{tp}^{(3)}$ and $\mathbf{PSN}^{(3)}$) remain aligned.

An example arbitrary configuration, and the transformations it undergoes are shown in Figure 4.12. The final projection of the scene is shown in Figure 4.13.

As with the two-dimensional case, matrices may be used to represent each transformation. The definition of each matrix is now considered.

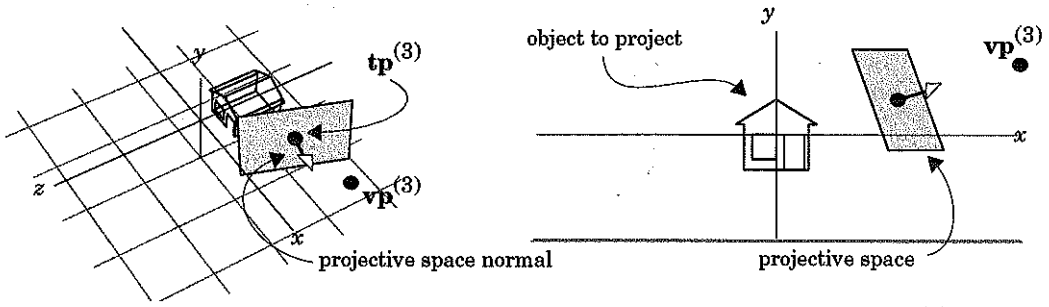
Translating the target-point to the origin. If a translation matrix is constructed with the negative of each component of **tp** as the amount to translate that component, that matrix will transform **tp** to the origin. Therefore, given $\mathbf{tp}^{(3)} = (tp_x, tp_y, tp_z)^T$, the matrix

$$\mathbf{T}^{(3)} = \begin{bmatrix} 1 & 0 & 0 & -tp_x \\ 0 & 1 & 0 & -tp_y \\ 0 & 0 & 1 & -tp_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

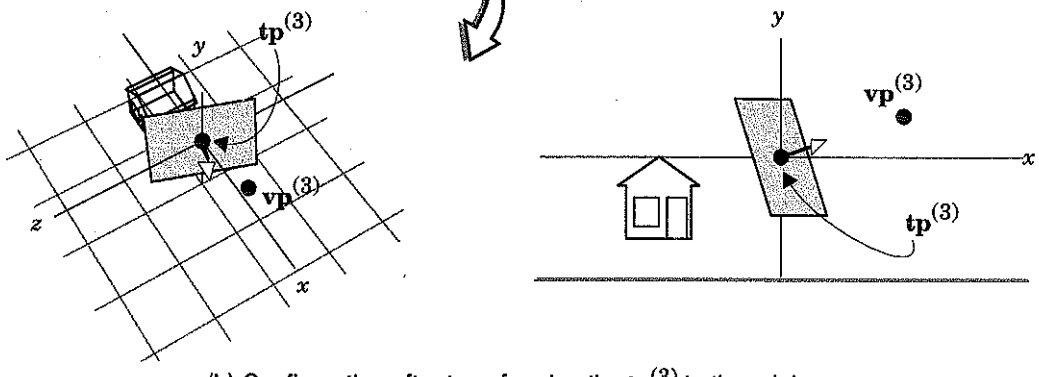
will translate the system so that an arbitrary target-point is at the origin.

Orienting the projective-space. There are at least two approaches to transforming an arbitrarily oriented projective-space to the constrained

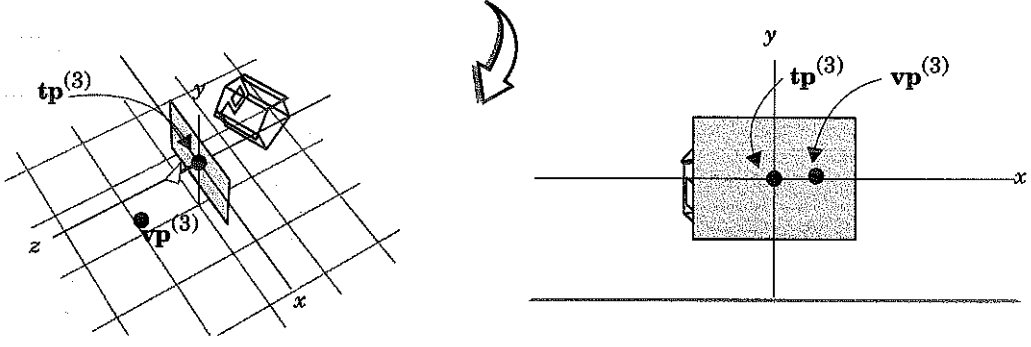
8. Rotation and shearing, as defined earlier, transform points about the origin, and hence are correctly defined when the centre of the system is at the origin.



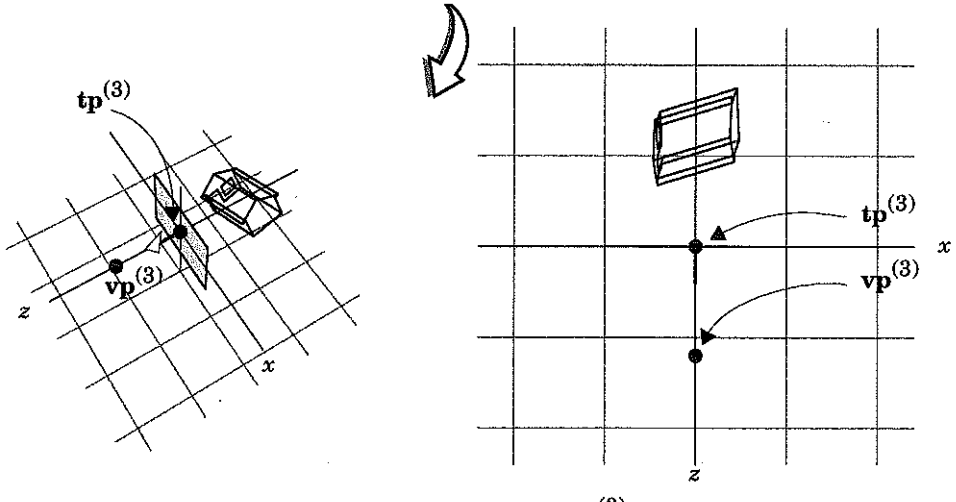
(a) Initial configuration. Note that the **psn** is not pointing directly toward the $\mathbf{vp}^{(3)}$.



(b) Configuration after transforming the $\mathbf{tp}^{(3)}$ to the origin.



(c) Configuration after the projective space has been oriented to be perpendicular to the *z*-axis and vertically aligned with the *y*-axis; $\mathbf{vp}^{(3)}$ is not aligned however.



(d) Configuration after shearing so that $\mathbf{vp}^{(3)}$ is on the *z*-axis; now suitable for projection.

Figure 4.12 Transformation of an Arbitrary Three-Dimensional View to the Projection-Ready Constrained View.

alignment necessary for projection: either use the properties of special orthogonal matrices, as used in the two-dimensional case; or, use a sequence of rotations to align each axis in turn. The special orthogonal matrix approach is adopted here, and an example of the latter approach is presented in Foley et al. (1991, pp. 218–220).

Recall that the unit row vectors of a special orthogonal matrix rotate into the principal axes of the coordinate system (Foley et al., 1991, p. 220). Two of the principal axes are known: the vertical direction of the projective-space, given by $\mathbf{psn}_1^{(3)}$, and the normal to the projective space, $\mathbf{psn}_2^{(3)}$. To orient the projective space to be normal to the z-axis, $\mathbf{psn}_2^{(3)}$ needs to rotate to be aligned with the positive z-axis. By arranging the local axes of the projective space into rows of a matrix, the appropriate transformation is established. The unknown axis is perpendicular to the two known ones, and may be found by using the cross product operator (Foley et al., 1991, p. 220).

Thus, the matrix to rotate the projective-space into alignment is

$$\mathbf{R}^{(3)} = \begin{bmatrix} i_x & i_y & i_z & 0 \\ j_x & j_y & j_z & 0 \\ k_x & k_y & k_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

where **i**, **j** and **k** represent the local axes of the projective space. The vector **k** will rotate into the z-axis, and is therefore obtained by normalising[9] $\mathbf{psn}_2^{(3)}$,

$$\mathbf{k} = \frac{\mathbf{psn}_2^{(3)}}{\left\|\mathbf{psn}_2^{(3)}\right\|}.$$

The **i** vector will rotate into the x-axis, and is given by normalising the cross product of **k** (the z-axis) and $\mathbf{psn}_1^{(3)}$ (the vertical direction of the projective-space):

$$\mathbf{i} = \frac{\mathbf{psn}_1^{(3)} \times \mathbf{k}}{\left\|\mathbf{psn}_1^{(3)} \times \mathbf{k}\right\|}.$$

9. A *normalised* or *unit vector* is a vector with length equal to one, obtained by dividing the vector by its length, as given by the modulus operator.

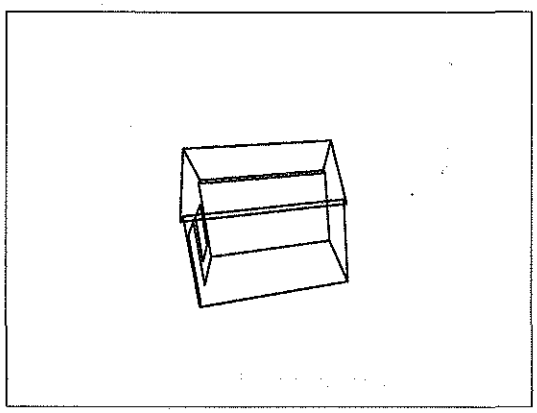


Figure 4.13 Final Projection of the Configuration Shown in Figure 4.12(a).

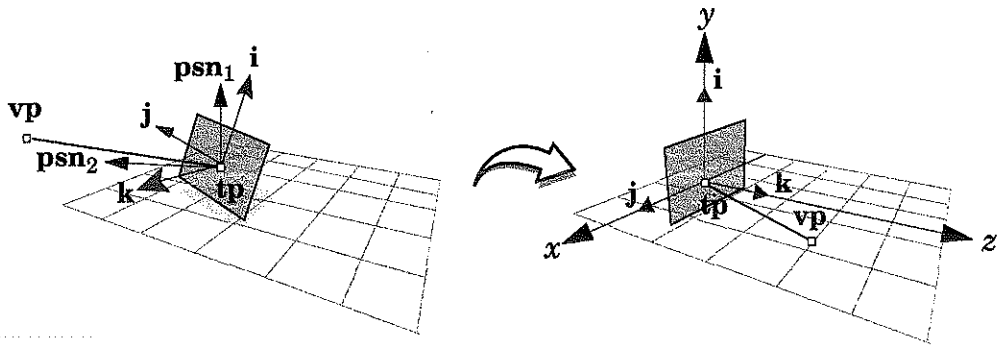


Figure 4.14 Orientation of a Three-Dimensional Projective-Space Before and After Transformation by $\mathbf{R}^{(3)}$ Matrix.

The direction that will rotate into the y-axis, $\mathbf{j}$, is given by taking the cross product of the other two axes,

$$\mathbf{j} = \mathbf{k} \times \mathbf{i}.$$

Normalisation of $\mathbf{j}$ is not necessary because the lengths of $\mathbf{k}$ and $\mathbf{i}$ are already one, so the cross product will also be of length one. Note that the cross product is not commutative, so $\mathbf{k} \times \mathbf{i}$ is different, in fact opposite, to $\mathbf{i} \times \mathbf{k}$. The order is chosen to ensure that the axes form a "right-handed" coordinate system (if the vectors were swapped, a left-handed system would result), and accords with the procedure defined by Foley et al. (1991, p. 261). Figure 4.14 shows a projective-space, with local axes labelled, before and after rotation by $\mathbf{R}^{(3)}$.

Shearing so the view-point is aligned. As may be seen in Figure 4.12(d), after the first two transformations, the projective-space and target-point meet the projection constraints, but the final constraint, that the view-point lie on the z-axis, is not yet fulfilled. If $\mathbf{vp}^{(3)\prime}$ is the so-far

transformed view-point[10], then by shearing along the x-axis to zeroise the x component of $\mathbf{vp}^{(3)'}$ and along the y-axis to zeroise the y-component, the $\mathbf{vp}^{(3)'}$ meets the projection constraint. The shear transformation matrix is therefore defined

$$\mathbf{Sh}^{(3)} = \begin{bmatrix} 1 & 0 & sh_x & 0 \\ 0 & 1 & sh_y & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where the shearing coefficients sh_x and sh_y are defined so that

$$\mathbf{Sh}^{(3)} \cdot \mathbf{vp}' = \begin{bmatrix} vp_x' - sh_x \cdot vp_z' \\ vp_y' - sh_y \cdot vp_z' \\ vp_z' \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ vp_z' \\ 1 \end{bmatrix}.$$

By rearranging this expression, the definitions for sh_x and sh_y are

$$sh_x = -\frac{vp_x'}{vp_z'} \quad \text{and} \quad sh_y = -\frac{vp_y'}{vp_z'}$$

where $\mathbf{vp}^{(3)'}$ is $\mathbf{vp}$ after transformation thus far, defined

$$\mathbf{vp}^{(3)'} = \begin{bmatrix} vp_x' \\ vp_y' \\ vp_z' \end{bmatrix} = \mathbf{R}^{(3)} \cdot \mathbf{T}^{(3)} \cdot \mathbf{vp}^{(3)}.$$

Because the shearing transformation also affects other points in the world-space, the projection of those points produces an image of the dataset, as if seen from the arbitrary view-point originally specified.

Combined three-dimensional viewing process. The three transformation matrices may be composed into a single matrix $\mathbf{M}_{\text{Arb}}^{(3)}$. The transformation from an arbitrary three-dimensional view to the projection-

10. Foley et al. (1991, p. 238) state that the view-point should be specified by the user in the already transformed state; this approach was primarily taken for use with parallel projections, and so is of no benefit for use with this study.

ready constrained view may be effected by multiplying all points in the world-space by $\mathbf{M}_{\text{Arb}}^{(3)}$. The matrix is defined

$$\mathbf{M}_{\text{Arb}}^{(3)} = \mathbf{Sh}^{(3)} \cdot \mathbf{R}^{(3)} \cdot \mathbf{T}^{(3)}. \tag{4.20}$$

4.3.3 Arbitrary Views in Four-Dimensions

Within this section less diagrams will be presented, principally because the concepts are identical to those introduced with two- and three-dimensional arbitrary view scenarios. Transformation of a four-dimensional system to conform to the projection-ready constraints follows the same three-stage process as with lower dimensions: namely,

- translate the system so the target-point $\mathbf{tp}^{(4)}$ is at the origin;
- orient the projective-space to be orthogonal to the x_4-axis and aligned with the principal axes of the system; and,
- shear the coordinate system so that the view-point $\mathbf{vp}^{(4)}$ is aligned with the x_4-axis.

The matrices representing the transformations at each stage are now examined.

Transform the target-point to the origin. This is directly analogous to lower-dimensions, and is represented by the matrix

$$\mathbf{T}^{(4)} = \begin{bmatrix} 1 & 0 & 0 & 0 & -tp_1 \\ 0 & 1 & 0 & 0 & -tp_2 \\ 0 & 0 & 1 & 0 & -tp_3 \\ 0 & 0 & 0 & 1 & -tp_4 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

where $\mathbf{tp}^{(4)} = (tp_1, tp_2, tp_3, tp_4)^{\mathrm{T}}$.

Orienting the projective space. The properties of special orthogonal matrices are again used to derive the matrix for aligning the local axes of the projective-space with the principal axes of the coordinate system. The difficulty, however, is in the nature of higher-dimensions: the

notions of both perpendicularity and rotation become more complicated, as Noll (1967, pp. 469–470) notes:

> ... in spaces of higher than three dimensions, rotation about an axis is meaningless ... since a multitude of non-parallel two-dimensional planes are all perpendicular to the same axis. For example, in four-dimensional space the x_1–x_2, x_1–x_3 and x_2–x_3 planes are all perpendicular to the x_4-axis.

Fortunately, the properties of special orthogonal matrices are not limited to lower dimensions—the unit row vectors of a special orthogonal matrix, independent of its dimensionality, rotate into the principal axes of a coordinate system (Foley et al., 1991, p. 207). Therefore, one method to orient the projective-space appropriately, is to create a special orthogonal matrix with the axes of the projective space as the row vectors of the matrix.

That is, given an array of four-dimensional normals $\mathbf{PSN}^{(4)} = (\mathbf{psn}_1^{(4)}, \mathbf{psn}_2^{(4)}, \mathbf{psn}_3^{(4)}, \mathbf{psn}_4^{(4)})$, the rotation matrix to align each normal $\mathbf{psn}_i^{(4)}$ with the ith principal axis is obtained by making each row equal to $\mathbf{psn}_i^{(4)}$ and then by homogenising the matrix:

$$\mathbf{R}^{(4)} = \begin{bmatrix} \mathbf{psn}_{1,1}^{(4)} & \mathbf{psn}_{1,2}^{(4)} & \mathbf{psn}_{1,3}^{(4)} & \mathbf{psn}_{1,4}^{(4)} & 0 \\ \mathbf{psn}_{2,1}^{(4)} & \mathbf{psn}_{2,2}^{(4)} & \mathbf{psn}_{2,3}^{(4)} & \mathbf{psn}_{2,4}^{(4)} & 0 \\ \mathbf{psn}_{3,1}^{(4)} & \mathbf{psn}_{3,2}^{(4)} & \mathbf{psn}_{3,3}^{(4)} & \mathbf{psn}_{3,4}^{(4)} & 0 \\ \mathbf{psn}_{4,1}^{(4)} & \mathbf{psn}_{4,2}^{(4)} & \mathbf{psn}_{4,3}^{(4)} & \mathbf{psn}_{4,4}^{(4)} & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix},$$

Giving the user both the flexibility and power of determining the entire normal vector array is a departure from the approach taken in the two- and three-dimensional scenarios. In those cases, the final axis is calculated in a relatively straight-forward manner, but similar algorithms are not so easily available in higher dimensions. More convenient methods of establishing the orientation of the projective-space are left to future research.

Shear the view-point into alignment. Directly analogous to lower-dimensional equivalents, the shear transformation is represented by the matrix

$$\mathbf{Sh}^{(4)} = \begin{bmatrix} 1 & 0 & 0 & sh_{x_1} & 0 \\ 0 & 1 & 0 & sh_{x_2} & 0 \\ 0 & 0 & 1 & sh_{x_3} & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

where the shearing coefficients sh_{x_i}, for $1 \leq i < 4$, are defined so the first three components of $\mathbf{vp}^{(4)}$ are zero. From previous discussion, the definition of sh_{x_i} is known to be

$$sh_{x_i} = -\frac{vp_{x_i}'}{vp_{x_4}}.$$

where $\mathbf{vp}^{(4)'}$ is the transformed **vp**, defined

$$\mathbf{vp}^{(4)'} = \begin{bmatrix} vp_{x_1}' \\ vp_{x_2}' \\ vp_{x_3}' \\ vp_{x_4}' \end{bmatrix} = \mathbf{R}^{(4)} \cdot \mathbf{T}^{(4)} \cdot \mathbf{vp}^{(4)}.$$

Combined four-dimensional viewing process. As before, the individual transformation matrices may be composed into a single matrix $\mathbf{M}_{\text{Arb}}^{(4)}$, which would effect each transformation. The matrix is defined as

$$\mathbf{M}_{\text{Arb}}^{(4)} = \mathbf{Sh}^{(4)} \cdot \mathbf{R}^{(4)} \cdot \mathbf{T}^{(4)}. \tag{4.21}$$

4.3.4 Arbitrary Views in n*-Dimensions*

The method for transforming a n-dimensional coordinate system, as described by n-dimensional viewing parameters, into the projection-ready constrained system, is a direct extension to the procedures for two-, three- and four-dimensions. Three individual transformations are required; these may be composed into a single transformation matrix.

Translate target-point to the origin. The target-point, defined $\mathbf{tp}^{(n)} = (tp_1, tp_2, \ldots tp_n)^{\mathrm{T}}$ is first translated to the origin using the matrix $\mathbf{T}^{(4)}$, where the elements, $T_{i,j}$ are defined:

$$T_{i,j} = \begin{cases} 1 & \text{if } i = j \\ tp_i & \text{if } i \neq j \text{ and } j = n+1 \\ 0 & \text{if } i \neq j \end{cases}$$

for $i, j = 1..n + 1$. Using this definition, $\mathbf{T}^{(2)}$, $\mathbf{T}^{(3)}$ and $\mathbf{T}^{(4)}$ are matrices for two-, three- and four-dimensional target-point translations, as discussed previously.

Orienting the projective-space. As noted in the Section 4.3.3, the orientation of the projective space and rotation become complicated in higher-dimensions. Further, the high-dimensional alternatives to the cross product, which offer "similar properties" (Goldman, 1982) but are not quite the same as the three-dimensional cross product, are complicated and beyond the scope of this project.

For these reasons, the following strategy is adopted for the specification of projective space orientation:

- the orientation of the projective space is given by a matrix $\mathbf{PSN}^{(n)} = (\mathbf{psn}_1^{(n)}, \mathbf{psn}_2^{(n)}, \ldots \mathbf{psn}_n^{(n)})$;
- each n-dimensional vector $\mathbf{psn}_i^{(n)}$ is a unit vector indicating the direction of the ith axis, local to the projective space;
- each direction vector is perpendicular to all others—that is, the matrix $\mathbf{PSN}^{(n)}$ forms an orthonormal basis in n-dimensions.

The matrix $\mathbf{R}^{(n)}$ to rotate the projective space into alignment is defined by placing each vector $\mathbf{psn}_i^{(n)}$ as a row, and then homogenising the matrix. In other words, $\mathbf{R}^{(n)}$ equals the homogenised version of the transpose of $\mathbf{PSN}^{(n)}$.

Shearing the view-point into alignment. From previous discussion, it may be shown that the shear matrix to align **vp** with the x_n axis is defined:

$$Sh_{i,j}^{(n)} = \begin{cases} sh_i & \text{if } i < n \text{ and } j = n \\ 1 & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases}$$

for $i, j = 1..n+1$. Using the definition above, $\mathbf{Sh}^{(2)}$, $\mathbf{Sh}^{(3)}$ and $\mathbf{Sh}^{(4)}$ are matrices for two-, three- and four-dimensional shear translations.

Combined *n*-dimensional viewing process. By composing the individual transformation matrices, a single matrix $\mathbf{M}_{\text{Arb}}^{(n)}$, defined as

$$\mathbf{M}_{\text{Arb}}^{(n)} = \mathbf{Sh}^{(n)} \cdot \mathbf{R}^{(n)} \cdot \mathbf{T}^{(n)}, \tag{4.22}$$

may be used to prepare an arbitrary n-dimensional configuration for projection, which may be repeated until the dataset is of a suitable dimensionality for output to a device.

4.4 Implementation

The implementation of the study seeks to verify the design of the dimensionally generalised rendering process, as presented so far this chapter, and demonstrate the effectiveness and potential of this method of high-dimensional visualisation. Firstly, the rationale behind the implementation is provided, followed by a brief but more in-depth look at the construction of the library.

4.4.1 Architecture Rationale

The preceding sections of this chapter have established processing requirements of the implementation which, in summary, is to provide a system allowing:

- submission and rendering of n-dimensional vertices, where $n > 0$;
- repeated projection of submitted vertices, from n-dimensions, optionally through an l-dimensional subspace, to k-dimensions, where $0 < k < l < n$; and,
- arbitrary views of the data through use of high-dimensional transformations.

In order to fulfil these requirements, the following facilities must also be available:

- facility to establish world-dimension (n), subspace dimension (l) and device-dimension (k);
- facility to establish arbitrary sequences of arbitrary transformations, at each of the world, subspace and device levels;
- facilities for pragmatically setting, and interacting with, the view parameters; and,
- a rendering architecture that integrates with an available window-system.

In deciding the programming model to adopt for the project, consideration was given to various approaches. However, the model provided by *Silicon Graphics' OpenGL* graphics library stands out as a base for the current implementation for the following reasons:

- **context-base or state-based processing:** within a program, a rendering context with various state variables (world-dimension, for example) is established. Access to the variables is only through *OpenGL* function calls, which are held together in a data-structure, enabling multiple contexts to be established;
- **transformation matrix stacks:** submitted vertices are multiplied by the matrix at the top of various matrix stacks. Stacks provide a dual advantage: firstly, separate stacks are maintained for different purposes (one for modelling transformations, one for projection[11]); and secondly, by pushing and popping matrices onto and from the stack, control is provided over the sequences of transformations, allowing them to be composed or isolated;
- **generalisable low-dimensional graphics facilities:** *OpenGL* has a wide range of functions for processing low-dimensional vertices. By generalising these functions, programmers familiar with *OpenGL* may move to this implementation with a minimum of confusion; and,

---

11. A third matrix stack is used for texture mapping in OpenGL, but is not relevant for high-dimensional use.

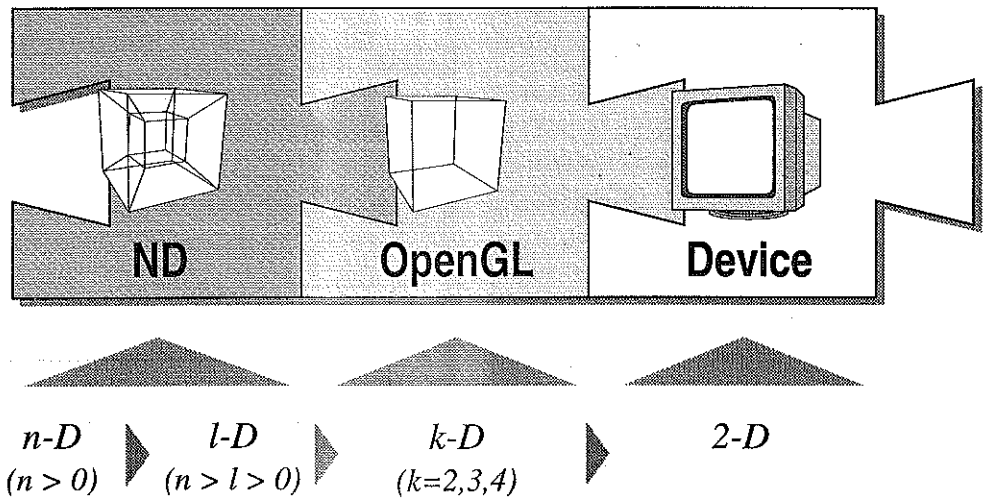


Figure 4.15 The *ND* Library in Relation to *OpenGL* and an Output Device.

- **integration aspects:** *OpenGL* is the library to be used for rendering processed vertices (see Chapter Three), because the architectures are compatible, the integration of the two is relatively seamless.

As noted above, OpenGL is used as a low-level graphics library for rendering the processed high-dimensional vertices. By using *OpenGL*, some aspects of this implementation are simplified: namely, a focus on only the transformation and projection of high-dimensional vertices is permissible, while technical aspects, such as rasterisation, colour, window creation and so on, may be omitted. The connection between this implementation, termed *ND* (for *n*-dimensional graphics library) henceforth, *OpenGL* and an output device is shown in Figure 4.15.

According, all functions follow *OpenGL*'s style and conventions. In addition, type names also follow the *OpenGL* naming convention, which prefixes names by "gl" or "GL", except that the *ND* library prefix is "nd" or "ND" respectively.

4.4.2 Design Specifics

The following four aspects of the design require brief elaboration: rendering contexts, transformation matrix stacks, vertex submission and arbitrary views.

Rendering contexts. Before projection of high-dimensional datasets may begin, the rendering context must be initialised appropriately. For example, the world- and device-dimensions must be established, and any initial transformations applied to the appropriate matrix stacks. Any relevant *OpenGL* states must also be initialised.

In practice, is a rendering context is data structure containing information pertinent to the current state of the visualisation system. The internals of the data structure are not available to the programme, but include the dimensionality of the world, for example, as well as other information. The matrix stacks (see below) are also stored in the context. Thus, multiple windows may be operated on simultaneously, each rendering with their own configuration.

The states that are available, together with the functions that access them, are shown in Table 1.

Transformation matrix stacks. High-dimensional analogues of OpenGL's matrix stacks are used at various places in the *ND* rendering process. Four stacks are provided:

- **World Projection Stack** and **World Modelview Stack**: conceptually, the projection stack is for camera transformations in the world-space (such as "move the camera up, re-orient the projective space" etc.) and the modelview stack is for transforming the dataset to be viewed in the world-space. Both are implemented identically, however, and are included for compatibility with *OpenGL*.
- **Subspace Projection Stack** and **Subspace Modelview Stack**: as above, except these stacks operate on the subspace projection, if used.

Several functions are available for setting and altering the matrix stacks. The following operations are available on stacks:

- **ndMatrixMode:** Determines which matrix stack is currently of focus;
- **ndLoadMatrix**: sets the top matrix to any arbitrary matrix;
- **ndLoadIdentity**: sets the top matrix to the identity matrix;
- **ndMultMatrix**: multiplies an arbitrary matrix with the matrix at the top of the current stack;

Table 1
Rendering Context States

State	Access Function(s)	Description
Dimensionality		
World_Dimension	ndDimension ndGet	The dimension of the world space; Initial dimensionality of vertices submitted for rendering.
Subspace_Dimension	ndDimension ndGet	Subspace projection is optional; but if used, this state indicates the dimensionality to project the world vector to, before being projected to the device.
Device_Dimension	ndDimension ndGet	Submitted vertices are projected to this dimensionality, and then passed on to OpenGL for rendering.
View Parameters		
Observer_Position	ndSet ndGet	The position of the observer along the *n*th axis; denoted as *r* in earlier discussion.
Hyperplane_Position	ndSet ndGet	The position of the hyperplane along the *n*th axis; denoted as *f* in earlier discussion.
Subspace_Projection_Hint	ndHint ndGet ndEnable ndDisable	Boolean indicating whether subspace projection is to be used.
Current vertices		
Raw_world_Vector	ndVertex ndGet	The most recently submitted vertex, before any transformation/projections.
Transformed_World_Vector	ndGet	This is the raw n-dimensional vertex, following transformations in the n-dimensional space, and prior to projection.
Raw_Subspace_Vector	ndGet	This is the raw *l*-dimensional vertex
Transformed_Subspace_Vector	ndGet	This is the transformed *l*-dimensional vertex, following subspace transformations, prior to projection.
Raw_Device_Vector	ndGet	The raw *k*-dimensional vertex; this is the vector submitted to OpenGL for rendering.
Transformation Matrix Stacks		
Current Stack	ndMatrixMode ndGet	Indicates which of the matrix stacks is currently of focus.
Current_Matrix	ndMatrixMode ndGet	The top matrix of the present matrix stack.

- **ndScale, ndTranslate, ndRotate**: the relevant transformation matrix is multiplied with the top of the current stack.

Vertex submission and rendering. Before projection of vertices may begin, a rendering context must be initialised appropriately. For example, the world- and device-dimensions must be established, and any initial transformations applied to the appropriate matrix stacks. Any relevant OpenGL states must also be initialised.

Submission of vertices using *OpenGL* is accomplished by a call to a function of the form:

```
glVertex<n>[v]<t>(...)
```

where n is two, three or four; "v", if present, indicates that the vertex is specified as an array, rather than a sequence of parameters; and "t" represents the type of the vertex— "f" meaning float, "i" meaning integer, and so on. Using *ND*, a n-dimensional vertex is submitted using a function of the form

```
ndVertexN[v]<t>(int n, ...)
```

where v and t are as before, and the parameter n indicates the dimensionality of the vertex to be rendered. In a sense, *OpenGL* may be considered the special case of *ND* where $n = 3$ and $k = 2$.

The stages a vertex undergoes prior to hand-over to *OpenGL* are shown in Figure 4.16. Code fragments, demonstrating the use of the *ND* library, are presented in Appendix B.

Practical approach to high-dimensional arbitrary view specification. As it may be clear from preceding discussion, the specification of an arbitrary view becomes complicated in four- and higher-dimensions. In particular, the orientation of the projective-space, as given by the array of normal vectors **PSN**, is difficult.

One practical approach to solve this problem is suitable for an interactive implementation: rather than specifying the view parameters and constructing the transformation matrices to effect the arbitrary view, the user may obtain an arbitrary view of the dataset by interactively translating, rotating and shearing the coordinate system in n-dimensional

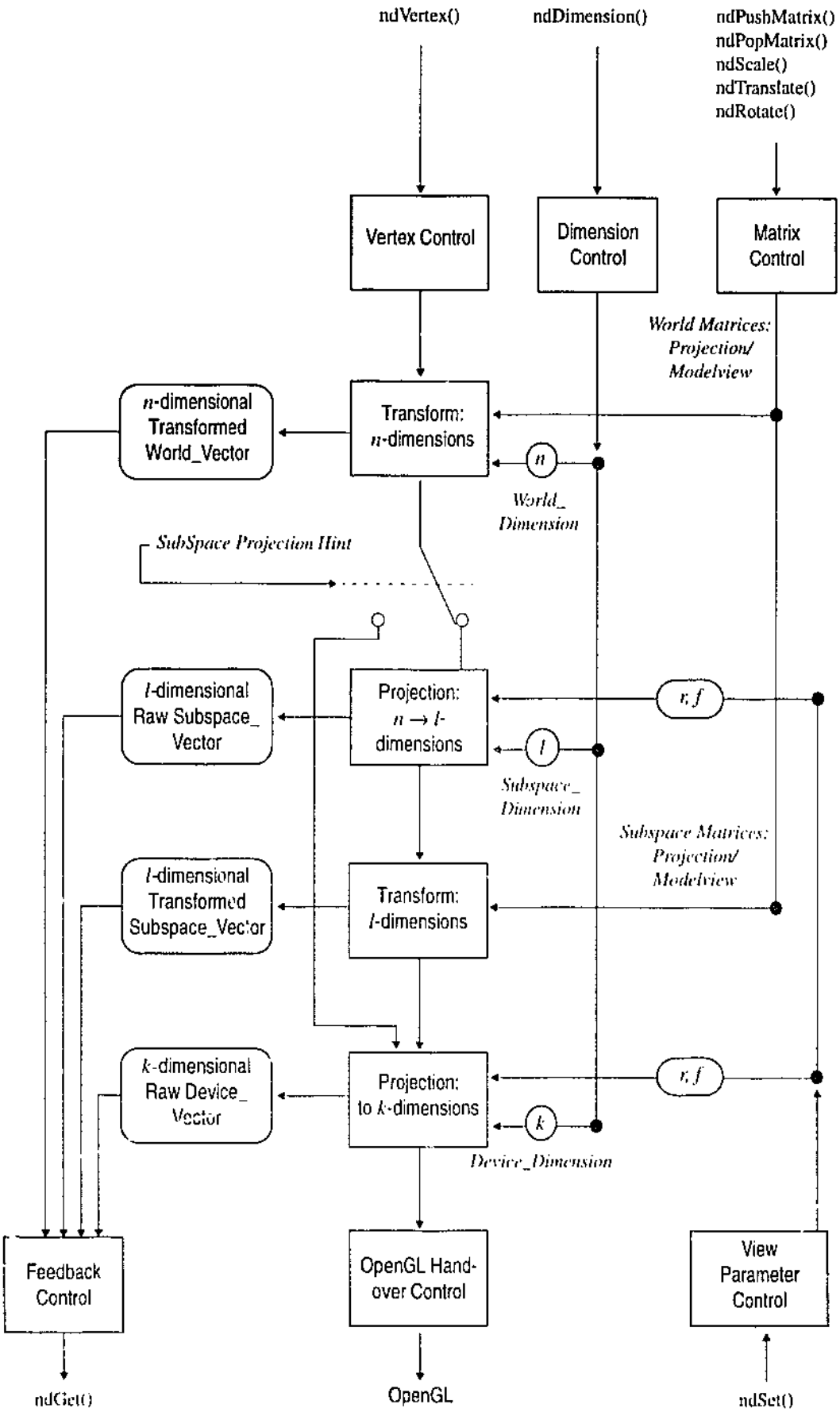


Figure 4.16 Schematic View of the *ND* Library.

space. The view parameters may, if necessary, be obtained by multiplying the projection-constrained parameters by the inverse of the transformation matrices. The view parameters that may be set pragmatically are f, the position along the nth axis to the projective space, and r, the position of the view-point along the nth axis.

Although this method of view specification may be unsuitable for applications, its use with this demonstration is appropriate because arbitrary views are still establishable, and interactively specified.

4.4.3 Summary of the Implementation

The implementation of the high-dimensional visualisation system has been designed so as to illustrate the use of the methods developed in this chapter, as an answer to the research questions posed in chapter two, and discussed in chapter three. The following chapter presents the results of the implementation.

5. Results

The high-dimensional visualisation methods developed throughout this study have been implemented in the form of a C function library named *ND*. Several programs, built using *ND* for high-dimensional processing and *OpenGL* for rendering the results of the processing, have been written specifically to demonstrate applications and the potential of the developed methods.

A gallery of images, with accompanying commentary, follows a brief explanation of the high-dimensional test objects and datasets.

5.1 The High-Dimensional Test Objects and Datasets

In order to demonstrate the projection and rendering of high-dimensional datasets, one or more such datasets must be defined. Four datasets have been chosen: namely, a n-dimensional hypercube, a four-dimensional *Klein bottle,* satellite remote-sensing data, and a geometric description of a car.

The Hypercube. Although various polytopes exist in higher dimensions (Banchoff, 1990; Coxeter, 1967), the hypercube is one which is not only defined in any dimension, but also relatively straightforward to render and understand. In two dimensions a hypercube is a plane; in three-dimensions a cube; and, in four-dimensions a cube extended perpendicular to itself in the next/fourth dimension (that is, depending on the view chosen, it may appear as a "cube-within-a-cube"). The progression of hypercubes for dimension zero to four is shown in Figure 5.1.

The module `polytope.c` provides functions for generating and storing the vertices and edges of a hypercube of any dimension. The vertices describing a unit n-dimensional hypercube $\mathbf{H}$, centred at the origin are defined with component values of ± 1 in all dimensions. A vertex $\mathbf{h}_i$ is connected to another vertex $\mathbf{h}_j$ by a line if and only if the distance between the vertices equals to the minimum possible distance (meaning that $\mathbf{h}_i$ and

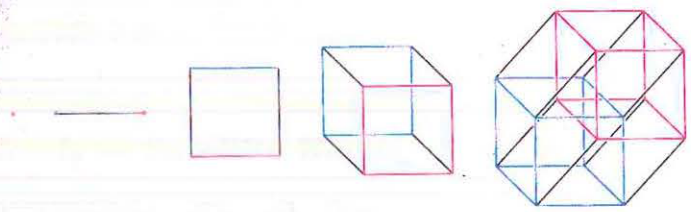


Figure 5.1 The Hypercube: Zero-Dimensional to Four-Dimensional. (From Banchoff, 1990, p. 9)

$\mathbf{h}_j$ differ in only one component), in this case, two. The connected line segments form surfaces, which may be rendered as solid polygons; but, for the purposes of this study, only a wire-frame representation of the cube is considered.

Many of the images of a hypercube are drawn with a thick line representing the surface which lies in a particular space; for example, some images of a four-dimensional hypercube show the edges joining vertices with no high-dimensional component drawn with thick lines, indicating the region of the hypercube that is "normal" three-dimensional space. Thin or dotted lines are used for all other faces.

Other high-dimensional polytopes exist, but with the exception of the Klein bottle (see below), their consideration is extraneous to the needs of this project.

The Klein bottle. This "extremely important" four-dimensional object is an example of a non-orientable surface, that can only be built in four-dimensions without self-intersection (Banchoff, 1991, p. 197). The vertices of a Klein bottle **H** are defined as

$$H(u,v) = \begin{bmatrix} (r + z\sin u)\cos v \\ (r + z\sin u)\sin v \\ (z\cos u)\cos\frac{v}{2} \\ (z\cos u)\sin\frac{v}{2} \end{bmatrix}$$

where the parameters u and v vary from 0 to 2π, in arbitrarily sized steps, and r is the constant radius of the bottle.[1] Coxeter (1969) and Banchoff

(1991) provide in-depth discussion about the background of the object and derivation of its definition.

Point-Clouds — Satellite Remote Sensing Data. Data obtained from real observations are, in practice, often the essentials for a visualisation package. Further, such datasets frequently describe a tuple space of dimension greater than three. Several remote-sensing datasets[2] were obtained and visualised using the *ND* library, each with various high-dimensionalities. The datasets contain measured observations over areas of land, with respect to certain variables. For example, the tuple-space configuration $\mathbf{d} = (x,\ y,$ red, green, blue, alpha) may represent a six-dimensional dataset yielding colour information for a point in a grid.

Geometric description of a car. By describing the surface of an object, such as a car with a series of polygons $\mathbf{P}_i$, a visual representation of the object may be constructed. Each $\mathbf{P}_i$ is in-turn described by v_i vertices, $\mathbf{P}_{i,j} = \left[x_j\ y_j\ z_j\right]^T$ for all j in v_i.

Although a car is only three-dimensional, inspection of the car post-processed in high-dimensions is a useful aid in understanding the high-dimensional visualisation system. In particular, as shown in Figure 5.5, to be discussed in Section 5.2.3, the three-dimensional car is a useful point-of-references as a high-dimensional system undergoes transformation.

---

1. Acknowledgement is made to Dr Mike Hartley, of the University of Western Australia, for his assistance with the definition of the Klein bottle.

2. The datasets were obtained from a public domain CD-ROM, produced by Earth Resource Mapper, Incorporated; acknowledgement is also given to David O'Brien, of Curtin University of Technology, for his assistance with the data.

5.2 The Gallery

Banchoff (1991, p. 11) notes that the use of computer graphics marks "a new era when it comes to visualizing dimensions." The images presented in this section endeavour to concur with Banchoff's statement and, with accompanying explanations, they are divided into various sections in order to address each of the requirements specified in chapter two: namely, to demonstrate

- **high-dimensional projection:** that is, projection from n- to $(n-1)$-dimensions;
- **repeated high-dimensional projection:** that is, projection from n- to k-dimensions, where $n > k > 0$;
- **arbitrary views:** that is, integration of view parameters and transformations facilitating arbitrary high-dimensional views;
- **visual enhancements:** that is, the integration of existing three-dimensional computer graphics image enhancement techniques, including shading, and hidden-surface elimination; and,
- **interactive facilities:** that is, the user facilities for interacting with the parameters/transformations controlling the rendering of the image.

Note that sequences of individual frames of an animation may only approximate the feeling or sensation of depth that may be gained by viewing graphics computer programs during execution.

5.2.1 High-Dimensional Projection

This section exemplifies the correctness of the design and implementation for performing projection from n- to $(n-1)$-dimensions.

Parts (a) and (b) of Figure 5.2 were rendered using *ND* to calculate the projection of a three-dimensional object onto a two-dimensional plane; i.e., $n = 3$ and $k = 2$. *OpenGL* is only used for mapping the two-dimensional primitives to the screen, not for any other transformation or projection processing. Images produced in this way may not allow *OpenGL* to perform any useful processing, such as depth-cued colours or surface rendering.

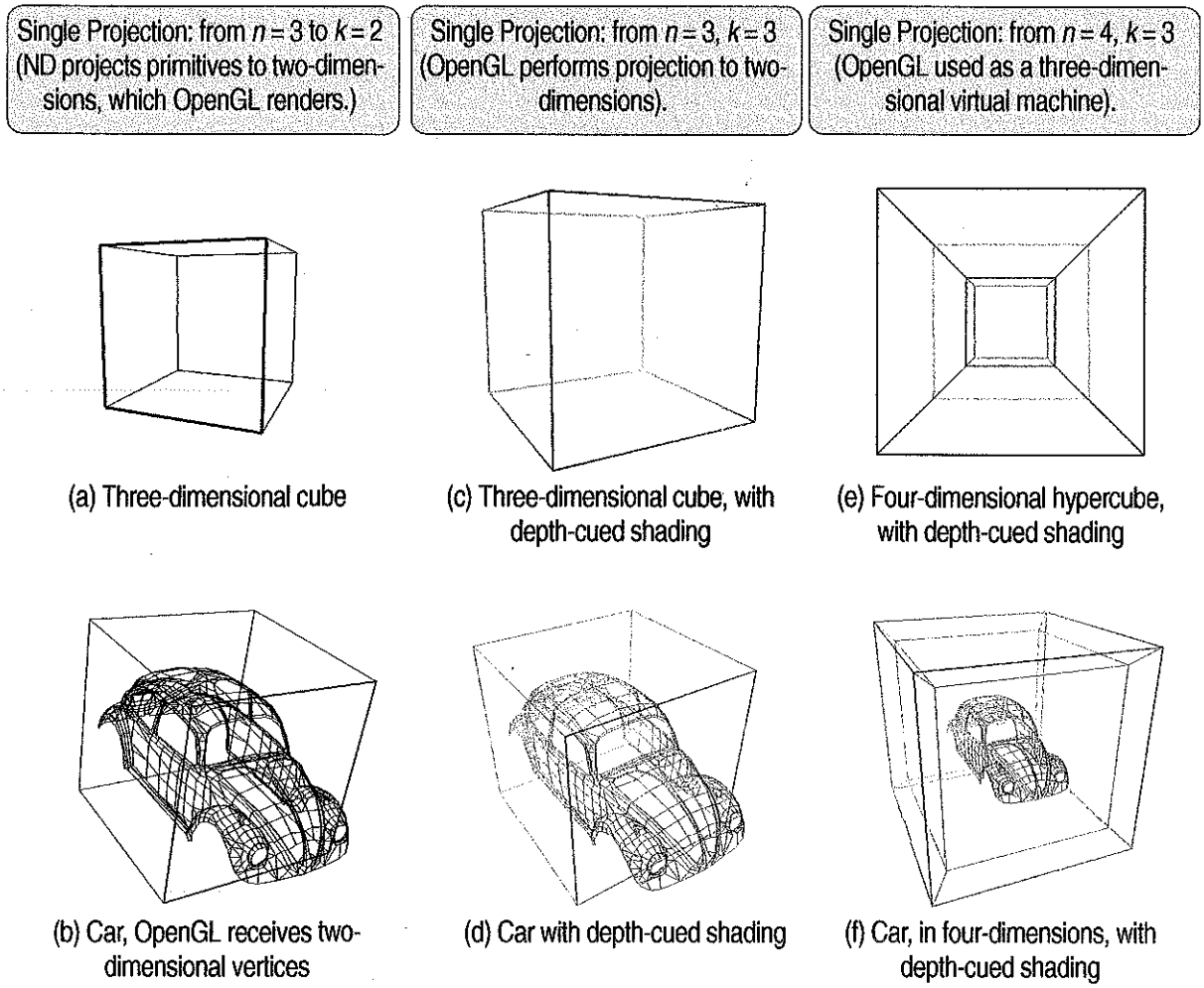


Figure 5.2 Images Demonstrating Single Projection: n- to $(n-1)$-dimensions.

Parts (c) and (d) of Figure 5.2 also show a single projection, except that $n = 3$ and $k = 3$ — that is, three-dimensional vertices have been submitted to *ND*, which has then passed them on to *OpenGL* for projection to two-dimensions and rendering. The important visual difference is that when *OpenGL* is given three-dimensional points to render, in-built depth cues, such as depth-based colour shading, may be incorporated. This depth cue is shown in parts (c)–(f) of Figure 5.2, where the lines are fainter, the "deeper" they are. Even this relatively simple depth cue adds considerably to the readability of images, therefore many images henceforth use $k = 3$, utilising the three-dimensional virtual machine provided by *OpenGL*.

Figure 5.2(e) and (f) demonstrate the final case of single projection: from four-dimensions to three-dimensions. Because $n = 4$ and $k = 3$,

OpenGL renders the three-dimensional projections of the four-dimensional world; in so doing, depth-cues may be incorporated if desired.

5.2.2 Repeated High-dimensional Projection

By repeating hyperplanar projection, a n-dimensional world space may be reduced to any k-dimensional subspace, where $n > k > 0$. Using *ND*, the rendering context (see Section 4.4) must be initialised with appropriate values for the world-, subspace- and device-dimensions. This is achieved by a code fragment such as:

```
int World_Dimension = n,
    Subspace_Dimension = 1,
    Device_Dimension = k;

int Subspace_Projection_Flag = FALSE;

ndDimension(ND_WORLD_DIMENSION,  World_Dimension);
ndDimension(ND_SUBSPACE_DIMENSION,  Subspace_Dimension);
ndDimension(ND_DEVICE_DIMENSION,  Device_Dimension);

ndBegin(GL_POLYGON);
    ndVertexNf(  4,   0.0,   0.0,   1.0,   1.0);
    ndVertexNf(  4,   0.0,   0.0, -1.0,   1.0);
    ndVertexNf(  4,   0.0,   0.0, -1.0, -1.0);
    ndVertexNf(  4,   0.0,   0.0,   1.0, -1.0);
ndEnd();
```

A rendering mode is then established with a call to `ndBegin()`, after which vertices are submitted and rendered appropriately. A call to `ndEnd()` indicates the completion of rendering or that a new rendering mode is required. For example, the code fragment above draws a unit plane in the $x_3\,x_4$ plane, within a four-dimensional world.

Within the call to functions of the form `ndVertexNf()`, the submitted vertex is projected according to the current dimensions and transformed by the top matrix of each of the matrix stacks.

Figure 5.4 shows the use of repeated projection to display images of four-, five- and six-dimensional hypercubes. An image of each hypercube in two states is provided: one without any transformation, and the other slightly rotated.

Repeated projection, in the case of hypercubes, is verified by considering the "cube-within-a-cube" appearance of subsequently higher-

dimensioned hypercubes. A four-dimensional cube is commonly acknowledged as appearing as a "cube-within-a-cube" prior to transformation in any high-dimension (Banchoff, 1991, p. 115; Noll, 1967, p. 471). The structure of a five-dimensional hypercube is more complicated: it appears as a cube-within-a-cube *within* a cube-within-a-cube. The topology of six- and higher-dimensional hypercubes continues this trend; not surprisingly, the structure of such objects is complex and difficult to understand.

Other data domains may also make use of repeated projection: Figure 5.3 shows a sequence of images depicting a subset of the same remote-sensing dataset, but with an increasing number of dimensions incorporated in the display. Part (a) demonstrates projection of only two variables, the (x, y) location of the sample; the display is a grid-like form. Part (b) shows the dataset with three-dimensions of each tuple included. A surface is now visible. Part (c) looks similar, except now four-dimensions are included so more transformations are possible. Part (d) illustrates the difference transformations can cause — the surface now appears as a comet-like point-cloud. Parts (e)–(g) use five dimensions of every tuple, repeating projection down to three-dimensions. The three frames illustrate different levels of transformation: (e) shows the surface-like initial appearance; (f) shows the appearance following some rotation of the structure — note some outliers (circled) become visible only under high-dimensional rotation; and, (g) shows the system after more extensive transformation, where the point-cloud appears as a pronouced comet-like structure.

Problem with repeated projection (revisited). Recall from Section 4.2.3 that repeated projection degenerates when $r = 0$ and $f = 1$ to a projection from $(k+1)$- to k-dimensions. This is shown in Figure 5.5, where a rotated five-dimensional hypercube is shown in a series of images with r approaching zero and f set to one, with $k = 3$. From the images, it is observable that as r approaches zero, the image becomes equivalent to projection from three- to two-dimensions, thereby confirming the observation stated in Section 4.2.3.

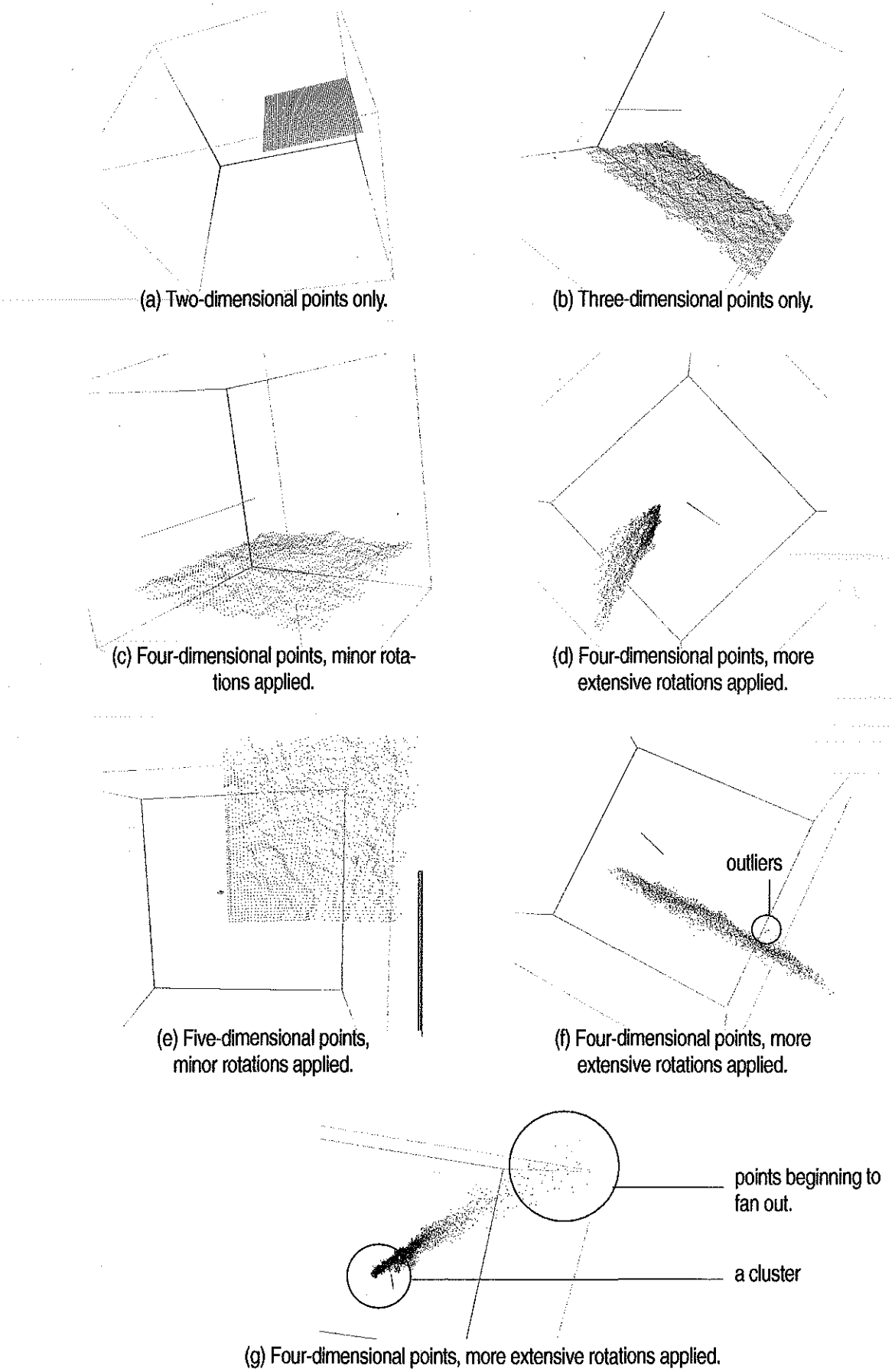


Figure 5.3 Demonstrating Repeated Projection: Incorporating More Dimensions Into a Visualisation of Remote-Sensing Data.

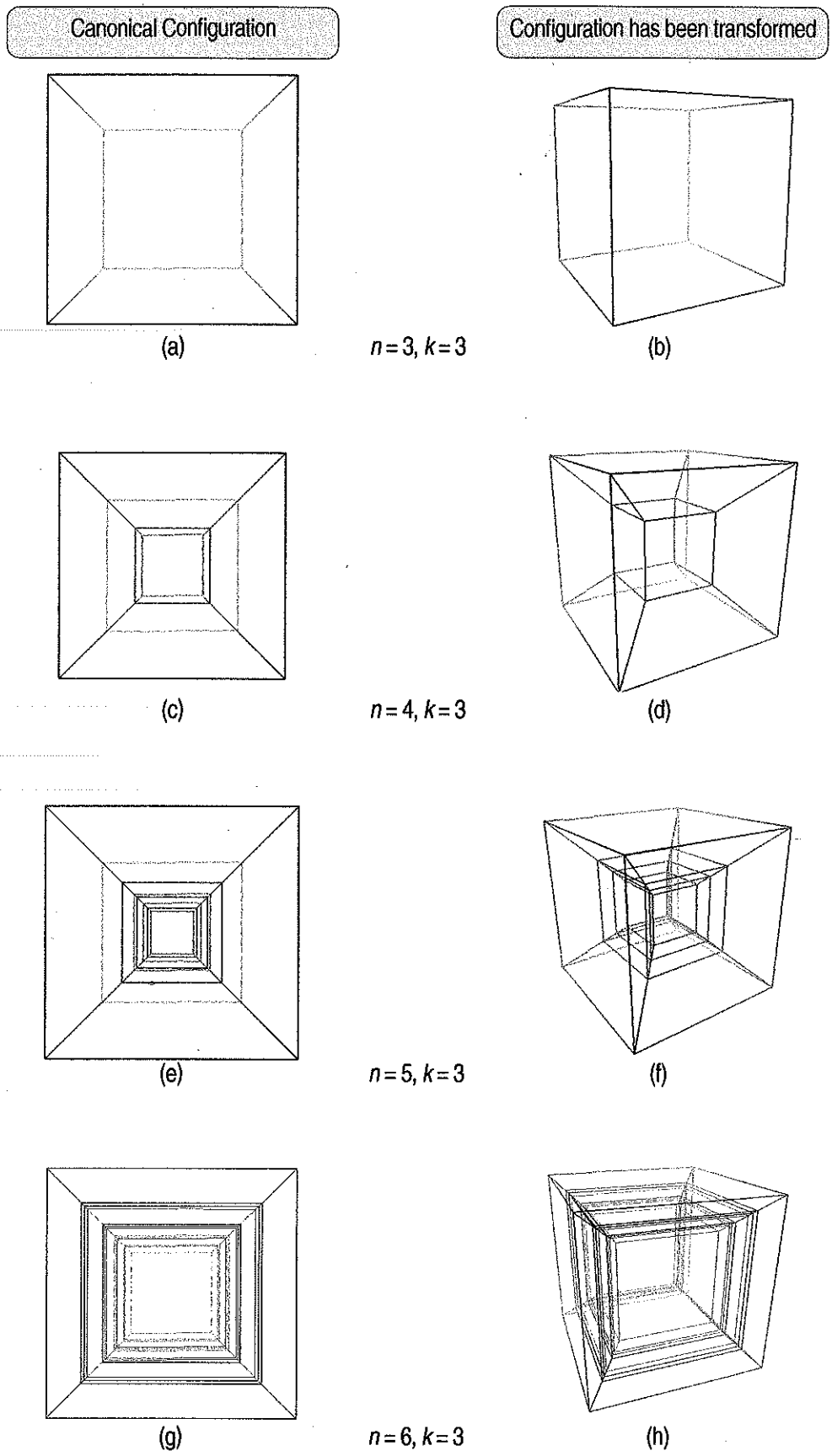
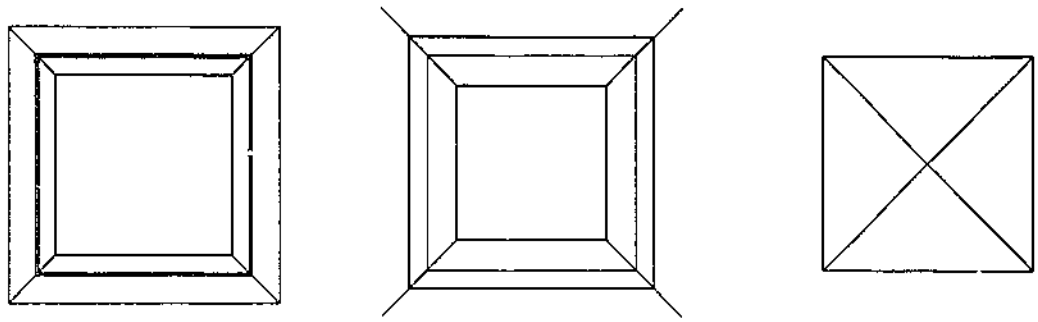


Figure 5.4 Hypercubes of Various Dimensionality, Demonstrating Repeated Projection.



(a) $r = -8, f = 1$ (b) $r = -2.7, f = 1$ (c) $r = 0, f = 1$

Note: $n = 4$, $k = 2$, therefore final image is planar (two-dimensional).

Figure 5.5 Degeneration of Repeated Projection.

5.2.3 Arbitrary Views: View Parameters and Transformations

As discussed in chapter four, arbitrary views of a scene may be achieved by using view parameters to define transformations which are applied to the coordinate system prior to projection.

Translation. By translating the coordinate system appropriately (see Section 4.3), *ND* will render images effectively focused on an arbitrary high-dimensional point. Parts (a) to (c) of Figure 5.5 show views of a car within a four-dimensional cube, obtained by moving the target-point in a line, as shown in part (d).

Rotation. The use of rotation transformations forms part of the process for specifying a view from an arbitrary point[3]. To be precise, rotation affects viewing from an arbitrary view-point, with the projective-space inclined to be perpendicular to the line from the view-point to the target-point. For example, Figure 5.5 shows a hypercube when viewed from various positions; each image is achieved by using rotations prior to projection.

---

3. All transformations, including rotation, are also commonly used for transforming only part of a scene.

The visual effect of rotating hypercubes should be noted. For example, consider how the faces — squares — of the three-dimensional cube appear to distort into trapezoidal shapes, then distort back to squares again, as the cube rotates. A similar effect occurs with the faces of four- and higher-dimensional hypercubes, as they undergo rotation, as shown in Figure 5.5.

Rotation in a high-dimensional plane produces images which are, at first, difficult to interpret: in Figure 5.5, the cube-within-a-cube appearance twists and distorts. These distortions occur because three-dimensional rotation causes the faces of a three-dimensional cube to become trapezoidal in appearance. Similarly, the surfaces of the four-dimensional hypercube — surfaces that are three-dimensional cubes — distort into trapezoidal forms that look like cut-off pyramids.

As rotation continues, cubes appear to return to reside inside each-other; the face rendered with a thicker line initially appears at the end of Figure 5.5(c), but squashes into the middle of the hypercube when rotated.

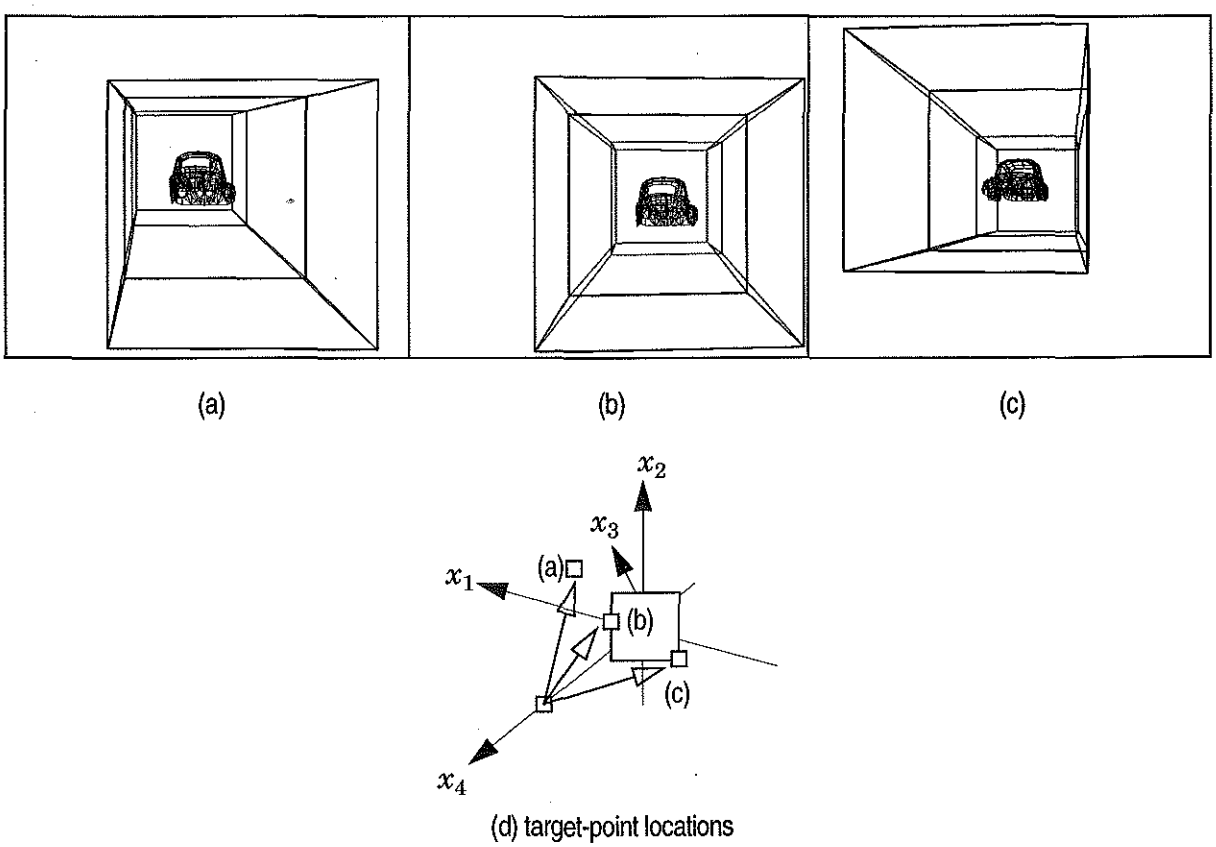


Figure 5.6 Successive Images, Using Translation, Demonstrating a Moving Target Point.

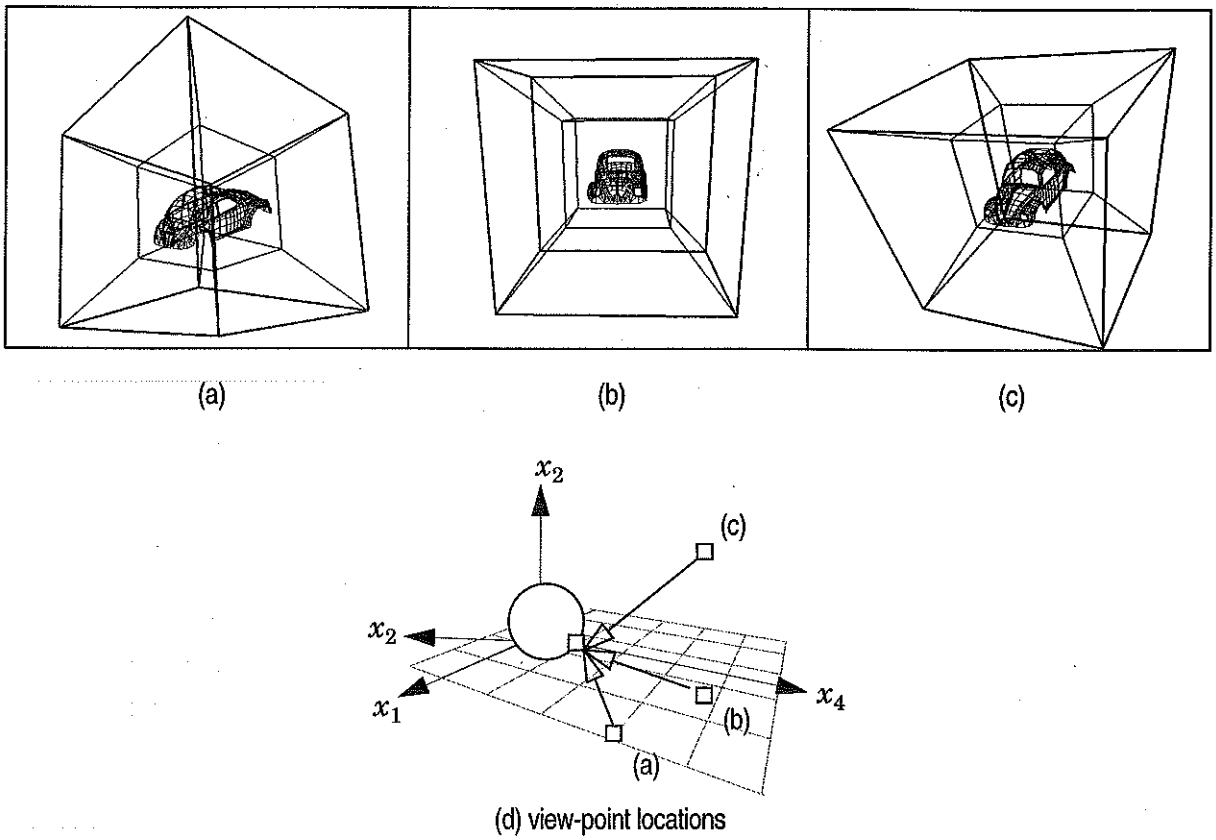


Figure 5.7 Successive Images, Using Rotation, Demonstrating a Moving View-Point.

This is analogous to the distortions of three-dimensional cubes under rotation: the front face of a three-dimensional cube (a plane), when viewed from the front, appears to enclose the back face which appears to be smaller. Subequently, when rotated appropriately, the back face of the cube appears to enclose the front. A car has been rendered inside the hypercube and it is useful as a point-of-reference as the hypercube undergoes rotation in different planes. Every vertex describing the car may be considered n-dimensional, with the last $(n-3)$ components equal to zero. Hence the face of the hypercube that the car is rendered inside is equivalent to "normal" three-dimensional space.

The usefulness of rotation, arbitrary points-of-view and projective spaces, is partly shown by Figure 5.5(c). Consider, for example, that the data displayed in the thickly drawn face are particular interesting. The configuration can be oriented so that these data appear in the centre of the hypervolume, or, alternatively, they may be squeezed off into a face if not of interest.

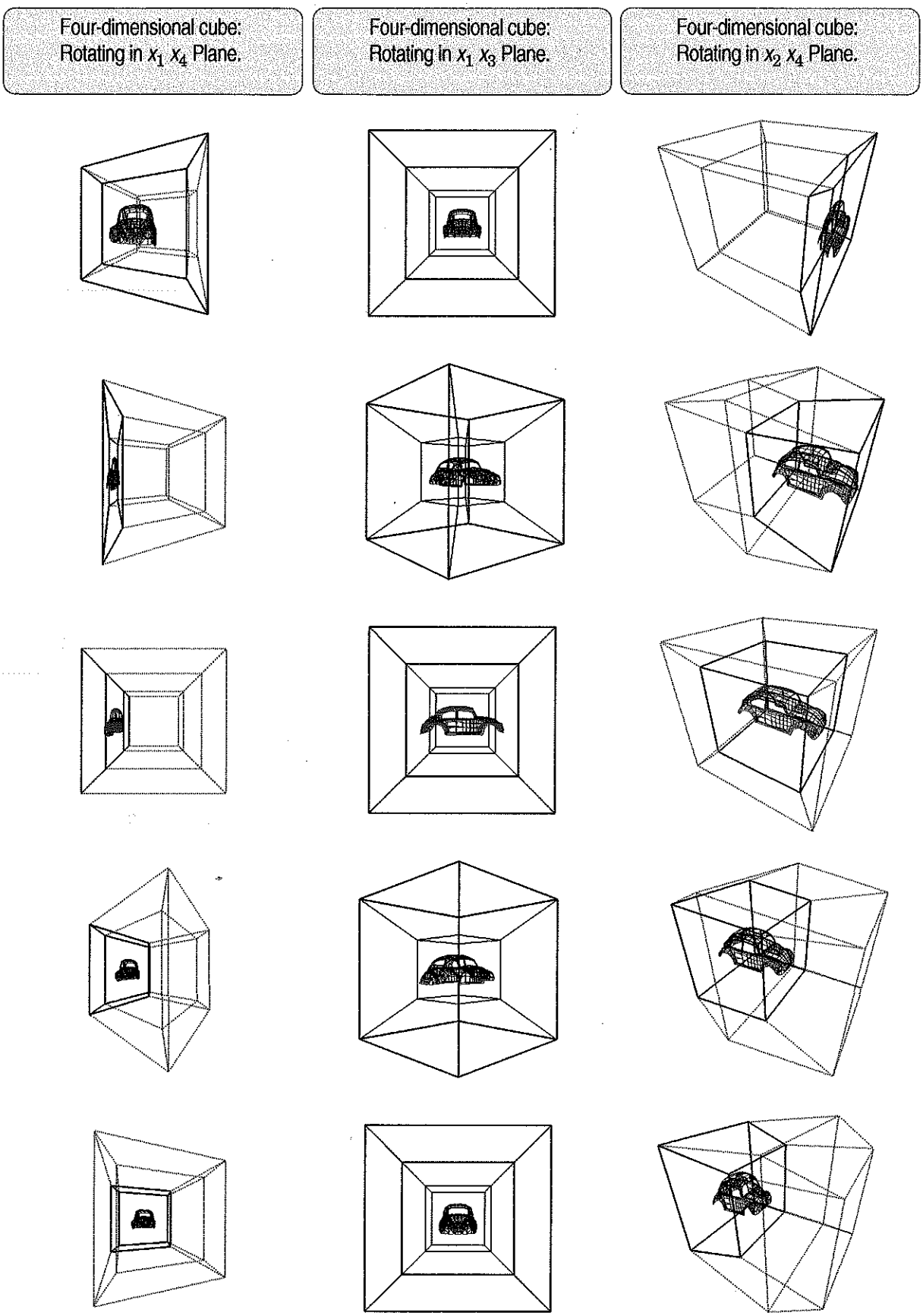


Figure 5.8 Successive Images of a Four-Dimensional Cube, Rotating in High-Dimensional Planes, Demonstrating a Moving View-Point.

The distortions of objects under rotation, as described above, are consistent with literature demonstrating high-dimensions projection, including Banchoff (1991), Koçak, Bissopp, Laidlaw, and Banchoff (1986), Hausmann and Seidel (1994) and Noll (1967).

Observer position and projective space position. Moving the observer and projective space positions, denoted r and f respectively, has a profound effect on the perspective foreshortening and exaggeration in resultant images, as shown in Figure 5.9.

The distance between r and f, given by $d = |r - f|$, determines the angle of the projectors from points in the world-space to the view-point. As d approaches infinity, the projectors approach parallel, and a parallel projection results. To make d large, either r or f must be large; however, in practice, the projective space (i.e., at $x_n = f$) is limited in its movement because it must lie reasonably close to the data to be projected or the image will become small.

In that case, giving r large values creates parallel projections, such as Figure 5.9(d) and (h); while allowing r to approach f creates perspective projections, of varying exaggeration, such as the other parts of Figure 5.9.

When $d \to 0$ (i.e. r approaches f), the projection becomes distorted and ultimately yields an empty image because no points are projected to the projective space. The relationship between projector angle and r and f is shown in Figure 5.10.

The effect of re-orienting the projective space. In three-dimensions, the orientation of the projective space (the "view-plane," according to Foley et al.) determines the variation of perspective or parallel projection that will result. When using a perspective projection, for example, the number of axes the projective-space cuts is the number of "vanishing points" visible in the projection (Foley et al., 1991, p. 231). That is, if the projective-space cuts only the z-axis (as with the constrained view), a single vanishing point is visible; and, if two axes are cut then two vanishing points are apparent, and so on.

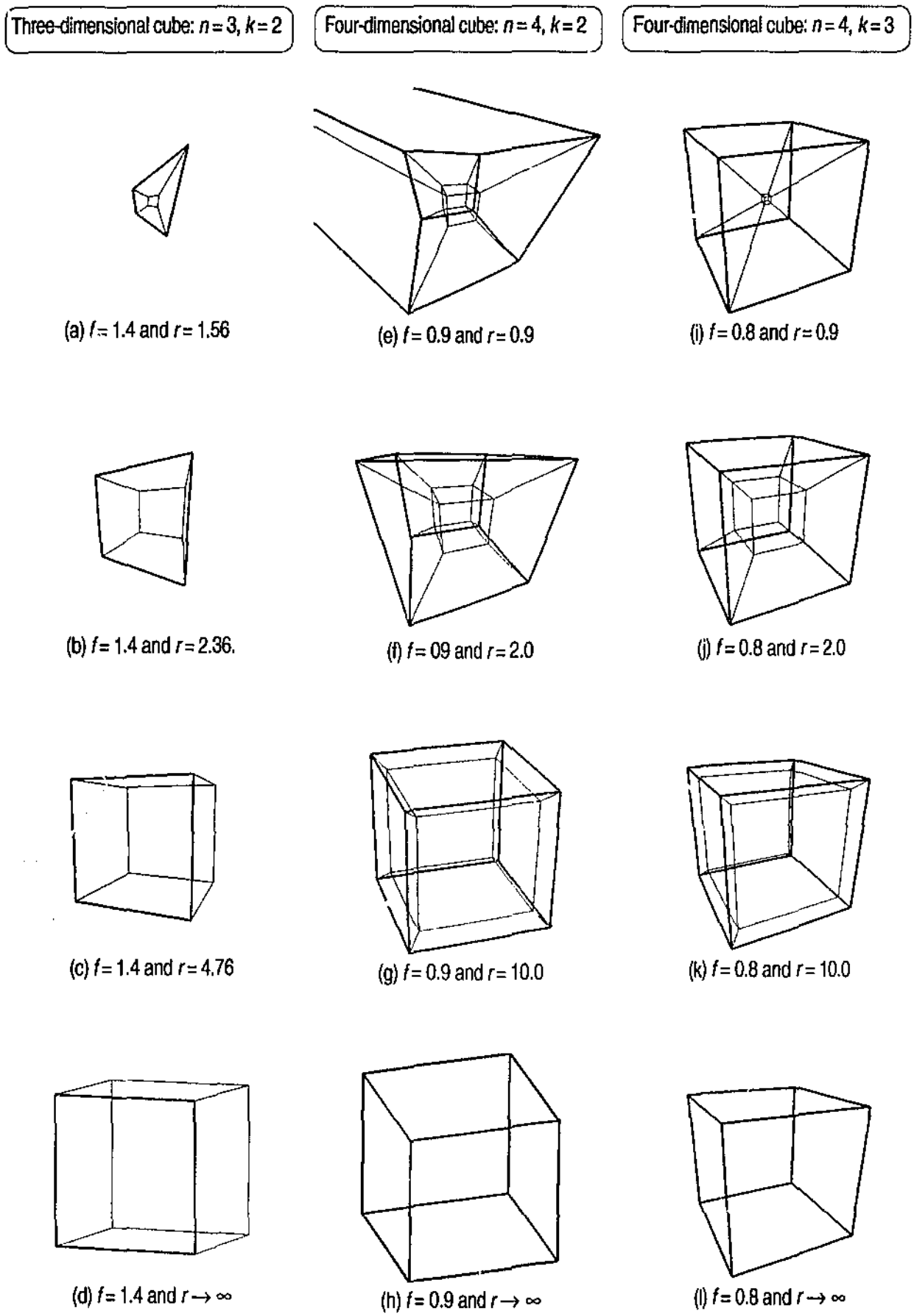


Figure 5.9 Varying Parameters r and f to Affect Perspective Distortion

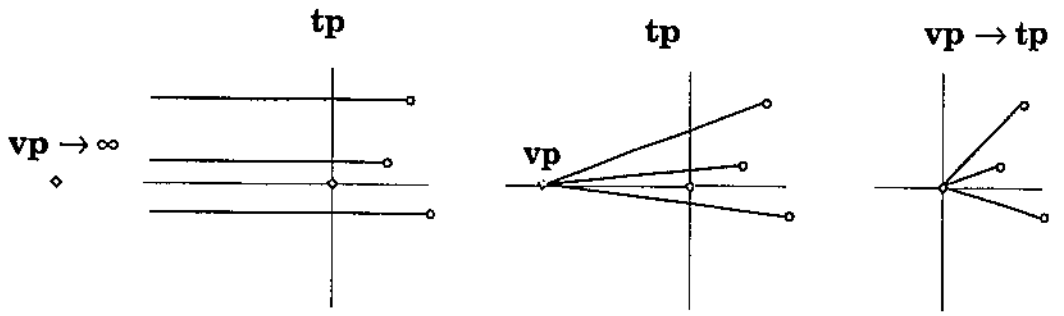


Figure 5.10 Angle of Projectors and the Distance Between r and f.

Figure 5.4(a) shows a one-point perspective of various hypercubes, while parts (c) and (h) of Figure 5.9 show three-dimensional, three-point perspective. Four-point perspective is shown in Figure 5.9(e)–(g) and (i)–(k), but it must be noted that the fourth vanishing point is the centre of the hypercube.

5.2.4 Visual Enhancements

Using *OpenGL*, some relatively straight-forward visual enhancements are possible: namely, arbitrary colouring, depth-based colour shading, and line thickness/pattern. Further features, such as surfaces and reflection, may be feasible, but would require considerable programming effort.

An alternative approach, not originally anticipated, is to use *Open Inventor*, a comprehensive object-oriented toolkit written "above" *OpenGL*, as the rendering component. Three programs was converted to make use of Inventor's features:

- Klein bottle visualisation: `nd/progs/inventor/klein.c++`
- hypercube visualisation: `nd/progs/inventor/hcube.c++`
- remote-sensing visualisation: `nd/progs/inventor/er1.c++`

Notably, only one of these programs, `klein.c++`, is complete enough for use.

In terms of implementation, the use of *Inventor* rather than *OpenGL* affords at least the following important advantages:

- surfaces may be rendered using features such as lighting, hidden-surface removal, reflection, and transparency, without the need for complicated calculations within *ND*; and,
- interaction, such as mouse dragging/sensing, may be linked to object selection/movement/rotation without the in-depth programming otherwise required.

Accordingly, the *ND/Inventor*-based programs demonstrate a higher level of realism and aesthetic appeal than their *OpenGL*-based relatives, as shown in Figure 5.11. Future development would be wise to use a toolkit such as *Inventor*, so that more of the full potential of *ND* may be realised.

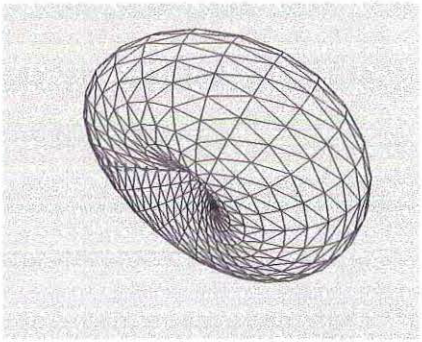
5.2.5 Interaction Facilities

All of the demonstration programs allow some interaction. Two directories of programs are available: firstly, the programs residing in the directory `nd/progs/gl`, use *OpenGL* for rendering, as was originally intended; secondly, the directory `nd/progs/inventor` houses several programs using *Open Inventor*.

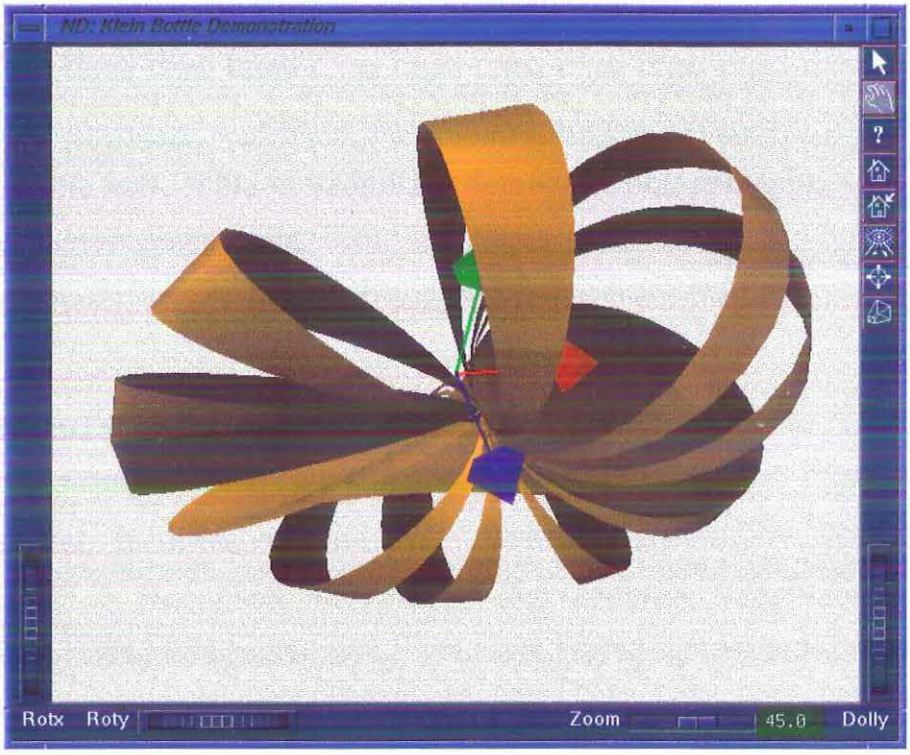
OpenGL-based programs. These programs have been used to produce most of the images depicted within this chapter. Control over the transformation/projection of the high-dimensional scene is afforded by a combination of mouse and keyboard controls. Specifically, the programs support the following user interaction:

- `er1.c`, `er2.c`: Mouse and keyboard control; and,
- `hcube.c`, `klein.c`, `vw.c`: Keyboard control only.

Dragging the mouse with the button down introduces a rotation, based on the position of the mouse, into the scene. Currently, mouse movement is linked only to low-dimensional transformation. Although mouse-based interaction with higher-dimensional rotation may be preferable, for



(a) A view of the Klein Bottle: $n = 4$, $k = 3$. Open Inventor removes hidden lines.



(b) Open Inventor's "Examiner Viewer" interface again; surface banding and axes is rendered using facilties on the interface

Figure 5.11 Example Images Produced by *ND/Inventor*-Based Programs.

demonstration purposes high-dimensional transformations are controlled by the keyboard. The available controls are:

- 1 – 6, a – f: set rotation plane; for example, typing "1" and then "c" sets the rotation to affect the x_1x_3 plane to;
- "\": switch between world-space transformation and subspace transformation;
- "|": toggle subspace projection on and off;

- space bar: rotates by a small amount in current plane;
- R, r: move the observer position r a positive or negative amount, respectively, along the nth axis;
- F, f: move the projective space position f a positive or negative amount, respectively, along the nth axis;
- S, s: scales larger and smaller respectively;
- T, t: translate by a positive or negative amount, respectively, along the current axis. The current axis is defined as the first component of the current rotation plane;
- H, h: shear by a positive or negative amount, respectively, along the current axis. This is useful for demonstrating arbitrary view-points/ projective-spaces; and,
- I: initialise; reset all transformations.

Open-Inventor demonstrations. The *ND/Inventor*-based programs use the interaction facilities supplied by *Inventor* together with the high-dimensional facilities of the *ND* library. Specifically, the following interaction is supported on each of the *Inventor* programs `er1.c++`, `hcube.c++`, and `klein.c++`:

- **The Inventor Examiner Viewer**, as shown in Figure 5.11, provides a range of functions for not only rendering, but for interaction with the view parameters and the rendered objects. For example, the mouse may be used to rotate, translate and scale the view, as well as for "selecting" objects by clicking (other operations might then be carried out on selected objects). Interface facilities also allow automatic "homing" to a point or object, display of coordinate-axes, and so on. Ideally, a new C++ class might inherit from Examiner Viewer and provide specific high-dimensional functionality; and,
- **Key Control:** The available controls are similar to those used in the *OpenGL*-based programs.

5.3 Summary of Results

The images presented throughout this chapter demonstrate that the algorithms developed in chapter four achieve the requirements of the project set out in chapter two: namely, to

- demonstrate single projection, from n- to $(n-1)$-dimensions;
- demonstrate repeated projection, from n- to k-dimensions, $n > k > 0$;
- incorporate transformations achieving arbitrary views of high-dimensional space and data;
- integrate visual enhancements; and,
- incorporate user interaction with view parameters/transformations.

Further, the images presented illustrate that useful information about the structure and distribution of the data depicted may be obtained through inspection.

6. Conclusion

6.1 Review

There is a "critical need" (LeBlanc, Ward & Wittels, 1990, p. 230) for computer visualisation tools which allow display and interpretation of high-dimensional datasets. Various dimensional reduction algorithms exist, as discussed in chapter three, allowing the dimensionality of a dataset to be reduced to a level where conventional visualisation algorithms may be applied and the output directed to a computer monitor or other device.

Three-dimensional planar projection, common in computer graphics, may be extended to perform projection from a n-dimensional space to a $(n\text{-}1)$-dimensional subspace. If this hyperplanar projection is repeated, a n-dimensional dataset may be reduced to two- or three-dimensions and rendered using existing computer graphics techniques.

This study has implemented repeated hyperplanar projection, and provided a software interface analogous to that of the *OpenGL* graphics library. The n-dimensional (*ND*) graphics library is a state-based environment, accepting tuples and transformations of any dimension. A tuple is first transformed in high-dimensional space — facilitating arbitrary views of the high-dimensional tuple-space — and then projected to k-dimensions, where k is any positive integer (commonly two or three). Optionally, a tuple may be projected to, and transformed within, an intermediate l-dimensional subspace before re-projection to a device. Rendering of the low-dimensional tuple is performed by *OpenGL*.

Several demonstration programs have been built using the implementation. Discussion of each program, with example output, is provided in chapter five.

6.2 Assessment

Overall assessment of the project is preceded by brief appraisal of the two main aspects of the project: namely, projection and arbitrary views.

Appraisal of projection. Noll (1967, p. 469) states that "no profound 'feeling' or insight into the fourth spatial dimension was obtained." It was anticipated that Noll's inability to find any "feeling" of higher-dimensions was because of the form of the animation he produced: a series of plots converted to a movie. It was therefore hoped that a computer-based interactive animation may yield greater spatial perception of higher-dimensions. This was found to be partially true — four-dimensional visualisations become reasonably intuitive to understand, but this was not the case for five- and higher-dimensional visualisations. More specifically, the following may be stated:

- the four-dimensional hypercube, after a while spent interacting with the view parameters, begins to exhibit predictable motion and distortion. That is, a glimmer of understanding of the fourth spatial dimension is achievable — an observation that is in stark difference to that of Noll's; and,
- unfortunately, perception of five- and higher-dimensional structure is difficult — the warping of the object when animated appears to give no intuitive indication as to its structure. These difficulties are due to several reasons: in all examples, the high-dimensional datasets/objects were projected to a device-dimension of three. Because of this, an effective three-dimensional illusion is essential to begin an understanding of the high-dimensional scenario. Such illusions require use of depth cues, as discussed in chapter three, and animation in order to be believable.

Interactive animation and colour depth cues (although extra to the requirements) were built into the example programs. By using even this relatively straight-forward depth cue, greater realism is achieved in the projected view of a three-dimensional space — but the realism is limited.

Other depth cues, for example occlusion (i.e. hidden-surface elimination) and surface shading, may assist a greater feeling of structure. However, such features were generally not incorporated in the examples built for this study (excepting the Klein bottle demonstration), because the

emphasis throughout has been on high-dimensional arbitrary views and projection.

Appraisal of arbitrary views. Arbitrary views of datasets and objects have proved invaluable for building an understanding of the distribution of the highly multivariate data and/or construction of high-dimensional objects.

In particular, view specification of the remote-sensing data proved useful — the structure and relationships within the dataset appear to warp and change; some outliers, clusters and trends are visible from some view-points, while some relationships become clear from other view-points.

Furthermore, the ability to interact with the views gives a user an inherent cognitive link between, say, the motion of their hand and the rotation displayed on screen.

For these reasons, it is the author's assessment that repeated high-dimensional projection, united with high-dimensional transformations, may offer computer-based visualisation industries a new and powerful approach to handle the visualisation of highly-multivariate data.

However, the lack of interface and analysis tools makes it difficult to perform any significant analysis of the visualised data. Consequently, further research is required into areas that may extend upon the framework established by this study.

Overall assessment. It is the author's opinion that repeated high-dimensional projection and transformations have proved to be successful and useful for rendering images of high-dimensional space and data.

Furthermore, the design and implementation of the n-dimensional visualisation system is sound and verified. Each of the research questions, as stated in chapter two, has been answered in the affirmative.

Finally, the visualisation method should ultimately be evaluated by experts in the field, who may make an assessment of the tool and perhaps recommend future directions for development.

6.3 Potential Future Research

Throughout the course of the project, several areas were noted as ones where the present research might be extended, and have been left to further research: namely,

- **greater control over repeated projection.** Currently, observer position and hyperplane position (r and f respectively) may be specified for the n-dimenional world space only. For every repeated projection, the same values of r and f are used. This may be both limiting and unnecessary; hence, future developments may allow r and f — and perhaps other transformations — to be specified at every dimension from n down to k;
- **high-dimensional clipping.** The present study facilitates n-dimensional transformations and repeated projection — no clipping of an arbitary high-dimensional hypervolume is performed. This means that, although arbitrary hypervolumes may be transformed and projected onto lower spaces, values outside of the desired volume are not "clipped" or removed, but instead may be unnecessarily rendered. By using high-dimensional clipping, values in specific ranges in each dimension may be included in a visualisation. However, high-dimensional clipping may significantly decrease performance;
- **greater interface facilities.** The *ND* library represents some part of the core functionality of a high-dimensional visualisation system. The integration of a set of interface tools and analysis utilities with *ND* may greatly assist a user's interpretation and exploration of their high-dimensional datasets;
- **transformation/projection of surface normal vectors.** *ND* is completely compatible with all of *OpenGL*'s rendering states. However, *OpenGL* provides functions for automatically calculating the shading of a surface based on its orientation and the placement of lights in a scene. In order to shade automatically a surface, the orientation of that surface, as given by normal vectors, must be included. *ND* does not allow for the submission (and subsequent transformation and projection) of normal vectors, because this would

be outside the requirements of the project.[1] By adding functions to allow submission of n-dimensional normal vectors, the description of a high-dimensional surface or object, including surface normals, may be projected to three-dimensional space;

- **combination of linear and abstract mapping.** Future high-dimensional visualisation systems may construct useful images of high-dimensional datasets by combining repeated hyperplanar projection with abstract forms of dimensional reduction;
- **industry-assisted development.** A differently focussed study, targetting visualisations of high-dimensional datasets that exhibit known properties and structure, may be of greater benefit in assessing the usefulness of this tool. Further, by incorporating expert opinion, directions for improvement and/or extension may become clear;
- **integration with other existing dimensional reduction methods.** Crawford and Fall (1990, p. 100) note that an "exhaustive search of high-dimensional space appears to be feasible for rather small (dimensions)." Perhaps the use of other dimensional reduction methods, including automated projection pursuit (Crawford and Fall, 1990), may yield usable images; and,
- **refinement and optimisation.** The implementation of *ND* and the demonstration programs sufficiently meet each of the requirements of the project. However, some optimisation and improvement to the code may improve the overall performance of the tool.

6.4 Conclusion

This project has investigated high-dimensional visualisation techniques, and established that there is a need for effective methods of visualising high-dimensional data. In answer to this need, the study has proposed and developed an alternative approach to existing techniques: namely, a dimensionally generalised projection technique, which may be incorporated with existing computer visualisation techniques.

---

1. Some function stubs, such as `ndNormalNF()` have been included in the source code, ready for expansion.

Several demonstration programs have been built to demonstrate the correctness and potential of the algorithms, and the results have been found to be effective and positive.

The development and implementation have satisfied each of the research questions, within the limitations of the study, and have highlighted areas where future research may be undertaken.

In summary, repeated high-dimensional projection, united with high-dimensional transformations, offers a new powerful effective approach to the visualisation of highly-multivariate data and/or objects.

References

Banchoff, T. F. (1990). *Beyond the Third Dimension: Geometry, Computer Graphics, and Higher Dimensions.* New York: Scientific American Library.

Beddow, J. (1990). Shape coding of multidimensional data on a microcomputer display. *Proceedings of the First IEEE Conference Visualization. Visualization '90* [CD-ROM]. Available IEEE IPO: Item: INSPEC 4169031 C9207-6130B-043. pp. 238–246.

Beshers, C., & Feiner, S. (July, 1993). AutoVisual: Rule-based design of interactive multivariate visualizations. *IEEE Computer Graphics and Applications, 13*(4), 41–49.

Calder, N. (1991). *Spaceship Earth.* Great Britain: Penguin.

Carlbom, I., & Paciorek, J. (1978, December). Planar geometric projections and viewing transformations. *Computing Surveys, 10*(4), 465–502.

Clifton, T. E., III, & Wefer, F. L. (1993). Direct volume display devices. *IEEE Computer Graphics and Applications. 13*(4), 57–65.

Cole, K. C. (1993, July). Escape from 3-D. *Discover*, 52–62.

Cox, D. J. (1988). Using the supercomputer to visualize higher dimensions: An artist's contribution to scientific visualization. *Leonardo, 21*(3), 233–242.

Coxeter, H. S. M. (1969). *Introduction to Geometry. 2nd Edition.* USA: John Wiley & Sons.

Crawford, S. L., & Fall, T. C. (1990). Projection pursuit techniques for visualizing high-dimensional data sets. In G. M. Nielson, B. Shriver (Eds.) & L. J. Rosenblum (Assoc. Ed.), *Visualization in Scientific Computing* (pp. 94-108). California: IEEE Computer Society Press.

Donoho, D. L., Huber, P. J., Ramos, E., Thoma, H. M. (1982). Kinematic display of multivariate data. In W. S. Cleveland, & M. E. McGill (1988), *Dynamic Graphics for Statistics* (pp. 111–120). CA: Wadsworth & Brooks/Cole.

Fisherkeller, M. A., Friedman, J. H., & Tukey, J. W. (1982). PRIM-9: An interactive multidimensional data display and analysis system. In W. S. Cleveland, & M. E. McGill (1988), *Dynamic Graphics for Statistics* (pp. 91–110). CA: Wadsworth & Brooks/Cole.

Foley, J. D., van Dam, A., Feiner, S. K., & Hughes, J. F. (1991). *Computer Graphics: Principles and Practice.* USA: Addison-Wesley.

van de Grind, W. A. (1986). Vision and the graphicssal simulation of spatial structure. Notes, based on presentation from *ACM 1986 Workshop on Interactive 3D Graphics.* Oct 23–24, 1986.

Haber R. N., & McNabb D. A.(1990). Visualization idions: A conceptual model for scientific visualization systems. In G. M. Nielson, B. Shriver (Eds.) & L. J. Rosenblum (Assoc. Ed.), *Visualization in Scientific Computing* (pp. 74-93). CA: IEEE Computer Society Press.

Hawking, S. W. (1988). *A Brief History of Time.* Australia: Transworld Publisher.

Hausmann, B. & Seidel, H-P. (1994). Visualization of regular polytopes in three and four dimensions. In M. Dæhlen and L. Kjelldahl (Eds.), *Eurographics '94. Computer Graphics Forum 13(3). Conference Issue. September 12–16, 1994.* C-305–C-316.

Hearn, D. & Baker, M. P. (1994). *Computer Graphics* (2nd Edition). New Jersey: Prentice Hall.

Hill, F. S. (1990). *Computer Graphics.* New York: Macmillan.

Kaufman, A. E., Nielson, G. M., & Rosenblum, L. J. (July, 1993). The visualization revolution. *IEEE Computer Graphics & Applications, 13*(4), 16.

Koçak, H., Bissopp, F., Laidlaw. D., & Banchoff, T. (1986). Topology and mechanics with computer graphics: Linear Hamiltonian systems in four dimensions. *Advances in Applied Mathematics, 7*, 282–308.

LeBlanc, J. Ward, M. O., & Wittels, N. (1990). Exploring n-dimensional databases. *Proceedings of the First IEEE Conference Visualization. Visualization '90* [CD-ROM]. Available IEEE IPO: Item: INSPEC 4169030 C9207-6160S-004.

Lindgren, C. E. S. (1978). Complex relations in urban and regional planning: An application of hypergraphics. In D. W. Brisson (Ed.), *Hypergraphics: Visualizing Complex Relationships in Art, Science and Technology*, (pp. 65–70). USA: Westview Press.

McDonald, J. A. (1983). Orion I: Interactive graphics for data analysis. In W. S. Cleveland, & M. E. McGill (1988), *Dynamic Graphics for Statistics* (pp. 179–199). CA: Wadsworth & Brooks/Cole.

Manning, H. P. (1921). *The Fourth Dimension Simply Explained.* Great Britain: Methuen & Co.

Mihalisin, T., Gawlinski, E., Timlin J., & Schwegler, J. (1990). Visualising a scalar field on an N-dimensional lattice. *Proceedings of the First IEEE Conference Visualization. Visualization '90* [CD-ROM]. Available IEEE IPO. pp. 255–262.

Nelson, T. R., & T. T. Elvins (1993). Visualization of 3D ultrasound data. *IEEE Computer Graphics and Applications.* Nov 1993. pp. 50–57.

Neider, J., Davis, T., & Woo, M. (1993). *OpenGL Programming Guide.* USA: Silicon Graphics.

Nielson, G. M., Shriver B. (Eds.), & Rosenblum L. J. (Assoc. Ed.) (1990). *Visualization in Scientific Computing*, USA: IEEE Computer Society Press.

Noll, A. M. (1967). A computer technique for display n-dimensional hyperobjects. *Communications of the ACM, 10*(8), 469–473.

Norton, A. (1982). Generation and display of geometric fractals in 3-D. *Proceedings of SIGGRAPH '82, Computer Graphics, 16*(3), 61–67

Pratico, C. J., Hanson, F. B., Xu, H. H., Jarvis, D. J. & Vetter, M. S. (1992). *Visualization for the Management of Renewable Resources in an Uncertain Environment.* Available CD: IEEE IPO CD-ROM.

Segal, M. & Akeley, K. (1994). *The Design of the OpenGL Graphics Interface.* Available by ftp://sgigate.sgi.com/pub/opengl/doc/design.ps.

Slaby, S. M. (1978). Geometry in applied science and engineering. In D. W. Brisson (Ed.), *Hypergraphics: Visualizing Complex Relationships in Art, Science and Technology*, (pp. 7–22). USA: Westview Press.

Sommerville, D. M. Y. (1958). *An Introduction to the Geometry of* n *Dimensions*. New York: Dover Publications.

van Walsum, T., & Post, F. H. (1994). *Selective visualization of vector fields*. New York: Dover Publications.

Watt, A. H. (1989). *Fundamentals of Three-Dimensional Computer Graphics*. Reading, Mass: Addison Wesley.

Wernecke, J. (1994). *The Inventor Mentor*. USA: Addison-Wesley.

Wolfram, S. (1991). *Mathematica: A System for Doing Mathematics By Computer. 2nd Edition*. CA: Addison-Wesley.

Appendix A. Implementation Structure and Source Code: *ND* Library

A.1 Overview

Within this appendix each source code file comprising the implementation of the *ND* library is listed and explained. There are two sections required:

- principal architecture: all files and functions directly relevant to the functionality as developed by this study. That is, all functions available to an external programmer making use of the ND library;
- underlying architecture: all files and functions used by the principal architecture—that is, mainly data structures and debugging;

Module for testing/verification of the library are listed in Appendix C, while programs making using the library to construct high-dimensional visualisations are included in Appendix B. The relationship between a third-party program using *ND*, the *ND* library itself, and the underlying structure of the *ND* library is shown in Figure 1.

A.1.1 Directory Structure

The *ND* library, test drivers and demonstration programs are kept in an archive. The directory hierarchy of this archive is shown in Figure 2.

A.1.2 Software Verification / Testing Framework

The *ND* library was designed to be modular for two purposes: firstly, although not a massive project, the library is not small, so modularised components helps to manage complexity; secondly, individual components may be tests as development continues. As each component is completed, it may be tested on its own, or with other (tested) components, depending on the particular area.

Three levels of debugging are incorporated into the implementation. The macro `Debug_Level` controls the amount of debugging

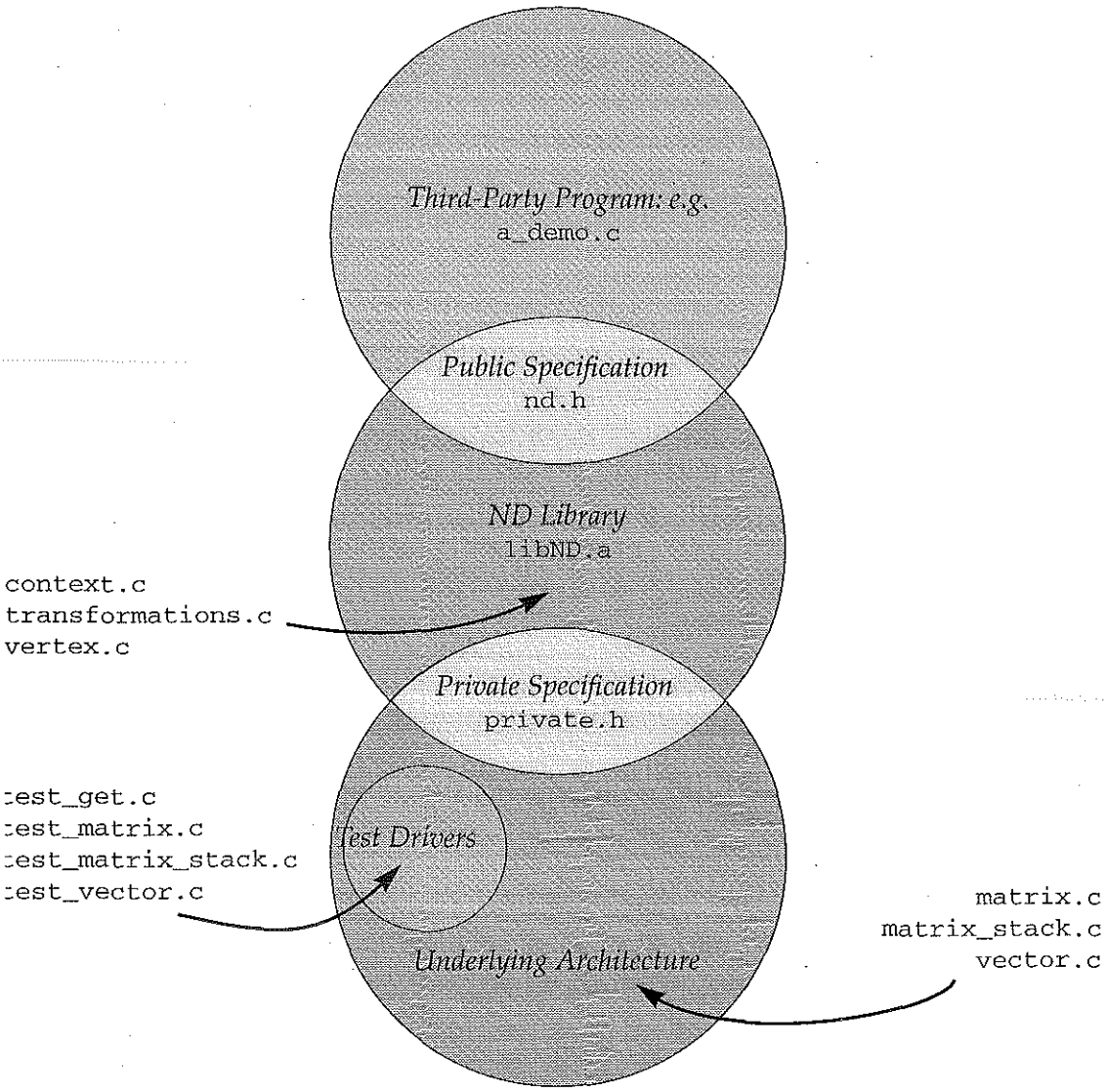


Figure A.1 Relationship and Visibility of Modules Within the Implementation, and with a Third-Party Program.

information that a compilation of the library incorporates. If `Debug_Level` equals the number 1, some debugging function calls are compiled; if `Debug_Level` equals the number 2, all debugging calls are compiled; and, if `Debug_Level` equals anything but 1 or 2, no debugging function calls are compiled.

Two further macros enable this multiple-level debugging facililty: `D(Any_Function_Call(...))` will compile to `Any_Function_Call(...)` if `Debug_Level` is at least one; and, `DD(Any_Function_Call(...))` will compile to `Any_Function_Call(...)` if and only if `Debug_Level` is two. By allowing multiple levels of debugging, some debugging information may be omitted entirely or partially, depending on the needs at the time. All

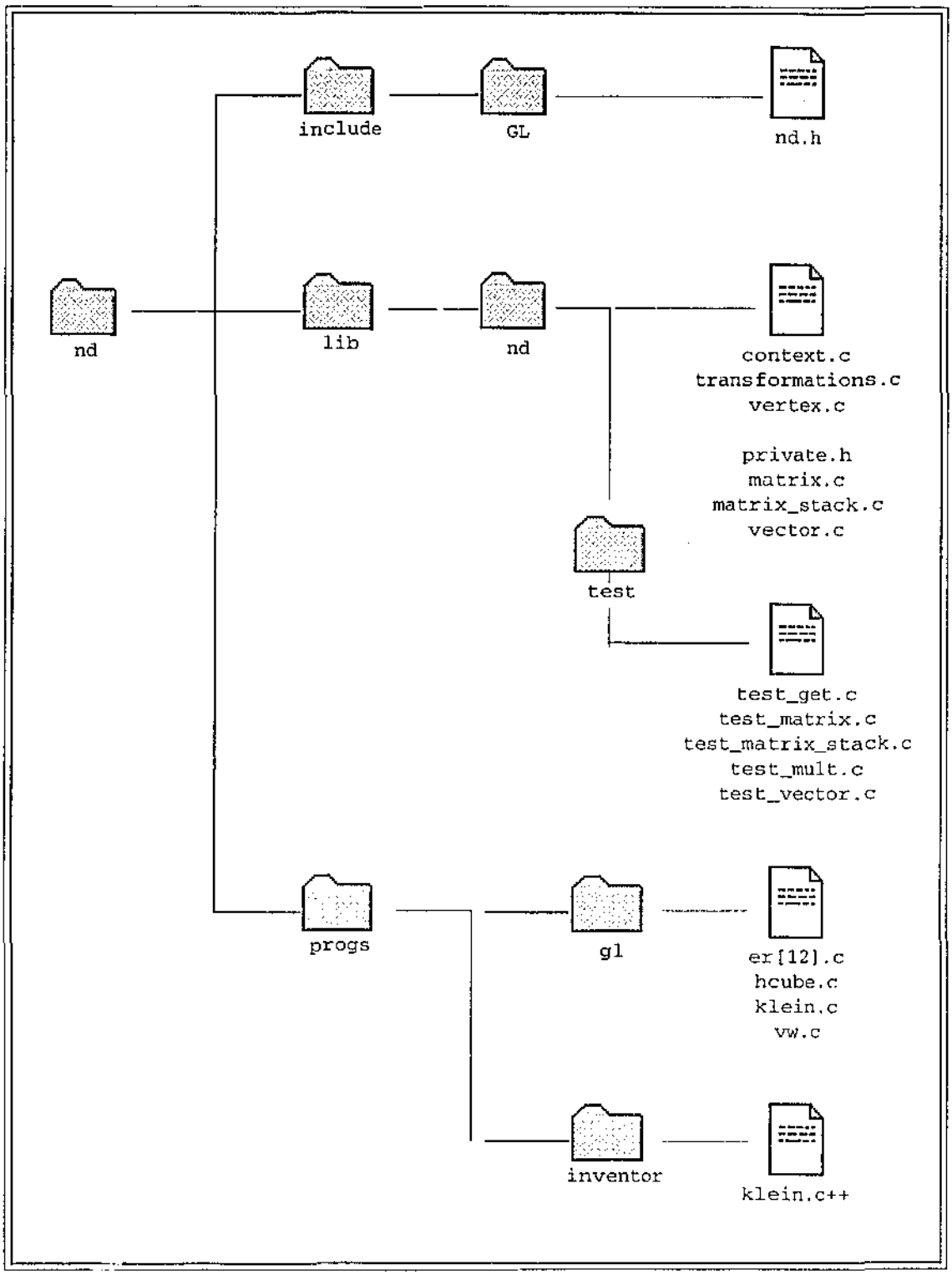


Figure A.2 Directory Hierarchy of ND Implementation.

debugging macros are defined in `private.h`.

Notably, a considerable number of debugging function calls are left in the source code. When `Debug_Level` is set appropriately, these calls are completely omitted from the compiled library, causing no loss in

performance.

A.1.3 Conventions

At the start of every ND library or component file, the following header appears:

```
/
*****************************************************************************
 *
 *           *         n-dimensional graphics library
 *************          ------------------------------
 *************
            *          A. Ellerton
            **         Project for part of BSc(Comp.Sci) Honours
*************           Edith Cowan University
************            (c) A. Ellerton 1995
*
*   *****               Component: <Filename>
***********             Description: <Description>
***       **
*          **
 *         *
*********************
*********************
*
*

*****************************************************************************/
```

where <Filename> is the name of the specific file, and <Description> is a brief description of the purpose of the code in the file.

A.2 Principal Architecture Source Code

The principal architecture source code is divided into three parts:

- *context handling:* where rendering context-specific settings, including world- and projective-dimensions may be established and altered;
- *matrix transformations:* where matrix manipulation allows the specification of arbitrary views and modelling transformations; and
- *vertex submission and projection:* where vertices of any dimension may be submitted to the library, projected to any lower dimension and rendered.

Each file, in each section, is listed alphabetically, following a listing of the *ND* library specification file.

A.2.1 Public Function Interface

The file `nd.h` is the C "include" file required by all programs intending

to use the *ND* library. Within this file, all available types, constants and functions are listed. The location of the file is in accordance with the convention set out by *OpenGL* — that is, the *ND* specification may be linked (via the `ln` command) to reside within the `/usr/include/GL` directory, together with the *OpenGL* specification.

nd/include/GL/nd.h

```
/*****************************************************************************
 *
 *
 *           *          n-dimensional graphics library
 *************          ------------------------------
 *************
            *          A. Ellerton
            **         Project for part of BSc(Comp.Sci) Honours
 *************         Edith Cowan University
 ************          (c) A. Ellerton 1995
 *
 *   *****
 ***********
 ***        **         Name: nd.h
 *           **        Description: All available ND functions/types etc.
  *          *
 *********************
 *********************
 *
 *

*****************************************************************************/

/*
 * Version:  0.1
 * Copyright (C) 1995 A.Ellerton
 *
 * Many thanks to Brian Paul (brianp@ssec.wisc.edu) for his Mesa GL/GLU
 * clone that has been SO useful during the development of this project.
 * Some of the form of (the implementation) of this library is based on
 * Brian's Mesa project - THANKS!
 */


#include <GL/gl.h>  /* For GL types */

#ifndef ND_H
#define ND_H


#ifdef __cplusplus
extern "C" {
#endif


#ifndef NULL
#  define NULL ((void *) 0)
#endif



/*
 * Apps can test for this symbol to do conditional compilation if needed.
 */
#define ND


typedef struct __NDXContext_Struct *NDXContext;

/*
```

```
 *
 * Enumerations
 *
 */

typedef enum {
/* Boolean values */
ND_FALSE = 0,
ND_TRUE = 1,

/* Data types */
        ND_BYTE = 100,
        ND_UNSIGNED_BYTE,
        ND_SHORT,
        ND_UNSIGNED_SHORT,
        ND_INT,
        ND_UNSIGNED_INT,
        ND_FLOAT,
        ND_2_BYTES,
        ND_3_BYTES,
        ND_4_BYTES,



/* Matrix Mode */
ND_MATRIX_MODE,
ND_WORLD_MODELVIEW,
ND_WORLD_PROJECTION,
ND_SUBSPACE_MODELVIEW,
ND_SUBSPACE_PROJECTION,

/* Render Mode */
ND_FEEDBACK,
ND_RENDER,
ND_SELECT,

/* Feedback */
ND_2D,
ND_3D,
        /* WHAT OTHERS? */

/* Buffers, Pixel Drawing/Reading */

        ND_WORLD_DIMENSION,
        ND_SUBSPACE_DIMENSION,
        ND_DEVICE_DIMENSION,

        /* States */
        ND_IGNORE_GL,                   /* ND pays no attention to GL */
        ND_LOOK_AFTER_GL,               /* ND takes note of GL */



/* Implementation limits */
ND_MAX_MODELVIEW_STACK_DEPTH,
ND_MAX_PROJECTION_STACK_DEPTH,
ND_MAX_CLIP_PLANES,
ND_MAX_VIEWPORT_DIMS,

/* Gets */
ND_CURRENT_NORMAL,
ND_MODELVIEW_MATRIX,
ND_MODELVIEW_STACK_DEPTH,
ND_PROJECTION_MATRIX,
ND_PROJECTION_STACK_DEPTH,
ND_RENDER_MODE,
ND_VIEWPORT,

ND_RAW_WORLD_VECTOR,
ND_TRANSFORMED_WORLD_VECTOR,
ND_RAW_SUBSPACE_VECTOR,
ND_TRANSFORMED_SUBSPACE_VECTOR,
ND_RAW_DEVICE_VECTOR,

ND_HYPERPLANE_POSITION,     /* PATCH 951213 */
```

```
ND_OBSERVER_POSITION,

/* Hints */
ND_USE_GL_HINT,

/* Utility */
ND_VENDOR,
ND_RENDERER,
ND_VERSION,
ND_EXTENSIONS,

/* Errors */
ND_INVALID_VALUE,
ND_INVALID_ENUM,
ND_INVALID_OPERATION,
ND_STACK_OVERFLOW,
ND_STACK_UNDERFLOW,
ND_OUT_OF_MEMORY,


        /* THE FOLLOWING SECTIONS OF OpenGL's ENUMERATIONS ARE EITHER NOT
           APPLICABLE TO ND OR HAVE NOT BEEN INCLUDING THE IMPLEMENTATION
           DUE TO OTHER CONSIDERATIONS SUCH AS TIME. */

/* Primitives */
/* Points */
/* Lines */
/* Polygons */
/* Display Lists */
/* Depth buffer */
/* Lighting */
/* User clipping planes */
/* Accumulation buffer */
/* Alpha testing */
/* Blending */
        /* Fog */
/* Logic Ops - OMITTED FROM THIS VERSION */
/* Stencil - NOT APPLICABLE */
/* Evaluators */
/* Scissor box */
/* Pixel Mode / Transfer */
/* Texture Mapping */
/* Extensions */

ND_ENUM_LAST
} NDenum;


/* ND_NO_ERROR must be zero */
#define ND_NO_ERROR GL_FALSE


/*
 *
 * Data types (architecture dependent in some cases)
 *
 */

/*  C typeGL typestorage                                */
/*--------------------------------------------------------------------------*/
typedef unsigned charNDboolean;




/*****************************************************************************
 * MATRIX STACK FUNCTIONS
 * ----------------------
 * Notes: No functions supporting double type floating point numbers have been
 *        included for simplicity - although the "f" suffix has been retained
 *        in keeping with OpenGL's style.

*****************************************************************************/
```

```
extern void ndLoadIdentity(void);
extern void ndLoadMatrixf(int N, const float *New_Matrix);
extern void ndMatrixMode(NDenum Mode);
extern void ndMultMatrixf(int N, const float *New_Matrix);
extern void ndRotate(int N, int A, int B, float Angle);
extern void ndScale(int N, double *V);
extern void ndTranslate(int N, double *V);
extern void ndIdentity();
extern void ndPopMatrix(void);
extern void ndPushMatrix(void);


/*****************************************************************************
 * PRIMITIVES

*****************************************************************************/
extern void ndVertexNf(int N, ...);
extern void ndVertexNd(int N, ...);

extern void ndNormalNfv(int N, float *V);
extern void ndVertexNfv(int N, float *V);
extern void ndVertexNdv(int N, double *V);

extern void ndBegin(GLenum Mode);
extern void ndEnd(void);


extern void ndDisable(NDenum State);
extern void ndEnable(NDenum State);
extern void ndHint(NDenum Target_Hint, NDenum Mode);
extern void ndRenderMode(NDenum New_Mode);
extern void ndDimension(NDenum Target_Space, int New_Dimension);
extern void ndSet(NDenum , float);
extern void ndGetfv(NDenum , float *);
extern void ndGetdv(NDenum , double *);

extern NDXContext ndXCreateContext(void);
extern void ndXMakeCurrent(NDXContext New_Current_Context);
extern void ndXDestroyContext(NDXContext A_Context);


/* ADD NOTHING AFTER THIS LINE */
#ifdef __cplusplus
}
#endif

#endif
```

A.2.2 Rendering Context Control

context.c

```
/*****************************************************************************
 *
 *
 *           *          n-dimensional graphics library
 ************          ------------------------------
 ************
            *          A. Ellerton
            **         Project for part of BSc(Comp.Sci) Honours
 ************          Edith Cowan University
```

```
 ************          (c) A. Ellerton 1995
 *
 *   *****             Component: context.c
 ***********           Description: All ND/GL context-related functions
 ***        **
 *           **
  *          *
 *********************
 *********************
 *
 *

*****************************************************************************/


#include <GL/nd.h>
#include "private.h"



/*****************************************************************************
 * TYPES/DEFINITONS ETC

*****************************************************************************/
static const DEFAULT_PROJECTION_STACK_SIZE = 2;
static const DEFAULT_MODELVIEW_STACK_SIZE = 8;

NDXContext Current_Context = NULL;



/*****************************************************************************
 * ND PUBLIC FUNCTIONS

*****************************************************************************/
```

void ndDimension(NDenum Target_Space, int New_Dimension)

```
/* Purpose: Set current context dimension appropriately.
 *
 */
{
    Check_Context_Exists("ndDimension");
    switch (Target_Space)
    {
        case ND_WORLD_DIMENSION:
            Current_Context->World_Dimension = New_Dimension;

            /* Setup vectors for this size worlds' dimensions.
               Notably, each vector is set to the world dimension; this
               is simpler, though potentially more expensive, because the
               upper componenets of (say) the device vector are used in the
               calculation of itself. Those upper components are ignored when
               rendering.  */
            New_Vector(New_Dimension+1,
                       &Current_Context->Raw_World_Vector);
            New_Vector(New_Dimension+1,
                       &Current_Context->Transformed_World_Vector);
            New_Vector(New_Dimension+1,
                       &Current_Context->Raw_Subspace_Vector);
            New_Vector(New_Dimension+1,
                       &Current_Context->Transformed_Subspace_Vector);
            New_Vector(New_Dimension+1,
                       &Current_Context->Raw_Device_Vector);

            /* Initialise matrix stacks to identity only */
            D(printf("ndDimension: \n\t"
                 "About to free World_Modelview_Stack\n"));
            Free_Matrix_Stack(Current_Context->World_Modelview_Stack);
            Push_Matrix(Current_Context->World_Modelview_Stack,
                        Identity_Matrix(New_Dimension+1));

            D(printf("ndDimension: \n\t"
                 "About to free World_Projection_Stack\n"));
            Free_Matrix_Stack(Current_Context->World_Projection_Stack);
            Push_Matrix(Current_Context->World_Projection_Stack,
```

```
                        Identity_Matrix(New_Dimension+1));

                D(printf("ndDimension: \n\tWorld is now %dD\n", New_Dimension));
                D(printf("\tContents of World_Modelview_Stack: \n"));

                D(Put_Matrix_Stack(Current_Context->World_Modelview_Stack));
                D(printf("\n\tWorld_Projection_Stack: \n"));

                D(Put_Matrix_Stack(Current_Context->World_Projection_Stack));
                break;

            case ND_SUBSPACE_DIMENSION:
                if (New_Dimension >= Current_Context->World_Dimension)
                {
                    fprintf(stderr, "ND: Cannot make subspace dimension %dD less "
                        "than or equal\n    to world dimension (%dD)\n",
                        New_Dimension, Current_Context->World_Dimension);
                }
                else
                {
                    Current_Context->Subspace_Dimension = New_Dimension;
                    D(printf("ndDimension: Subspace Dimension = %d\n",
                        New_Dimension));
                    Warning("ndDimension(): Subspace projection not yet "
                        "implemented\n");

                    /* Initialise matrix stacks to identity only */
                    Free_Matrix_Stack(Current_Context->Subspace_Modelview_Stack);
                    Push_Matrix(Current_Context->Subspace_Modelview_Stack,
                                Identity_Matrix(New_Dimension+1));

                    D(printf("ndDimension: \n\t"
                        "About to free Subspace_Projection_Stack\n"));
                    Free_Matrix_Stack(Current_Context->Subspace_Projection_Stack);
                    Push_Matrix(Current_Context->Subspace_Projection_Stack,
                                Identity_Matrix(New_Dimension+1));

                D(printf("ndDimension: \n\tSubspace is now %dD\n", New_Dimension));
                    D(printf("\tContents of Subspace_Modelview_Stack: \n"));

                   D(Put_Matrix_Stack(Current_Context->Subspace_Modelview_Stack));
                    D(printf("\n\tSubspace_Projection_Stack: \n"));

                  D(Put_Matrix_Stack(Current_Context->Subspace_Projection_Stack));
                }
                break;

            case ND_DEVICE_DIMENSION:
                if (New_Dimension < 2 || New_Dimension > 4)
                {
                   Warning("Cannot set device dimension anything but 2,3 or 4\n");
                    return;
                }
                Current_Context->Device_Dimension = New_Dimension;
                Current_Context->OpenGL_Draw_Function =
                    (New_Dimension == 3)? glVertex3dv : /* most likely */
                    (New_Dimension == 2)? glVertex2dv :
                                          glVertex4dv;
                D(printf("ndDimension: Device is now %dD\n", New_Dimension));
                break;

            default:
                break;
        }
    }
```

void ndDisable(NDenum State)

```
{
    Check_Context_Exists("ndDisable");
    switch (State)
    {
        case ND_SUBSPACE_PROJECTION:
            Current_Context->Subspace_Projection_Hint = ND_FALSE;
            D(printf("ndDisable: ND_SUBSPACE_PROJECTION \n"));
            break;
```

```
        default:
            break;
    }
}
```

void ndEnable(NDenum State)

```
{
    Check_Context_Exists("ndEnable");
    switch (State)
    {
        case ND_SUBSPACE_PROJECTION:
            Current_Context->Subspace_Projection_Hint = ND_TRUE;
            D(printf("ndEnable: ND_SUBSPACE_PROJECTION \n"));
            break;

        default:
            break;
    }
}
```

void ndRenderMode(NDenum New_Mode)

```
{
    Check_Context_Exists("ndRenderMode");
    switch (New_Mode)
    {
        case ND_FEEDBACK:
            Current_Context->Render_Mode = New_Mode;
            break;

        case ND_RENDER:
            Current_Context->Render_Mode = New_Mode;
            break;

        case ND_SELECT:
            Warning("ndRenderMode: Selection Mode not yet supported\n");
            break;

        default:
            break;
    }
}
```

void ndHint(NDenum Target_Hint, NDenum Mode)

```
{
    switch (Target_Hint)
    {
        case ND_SUBSPACE_PROJECTION:
            Current_Context->Subspace_Projection_Hint = Mode;
            D(printf("ndHint: Set hint ND_SUBSPACE_PROJECTION to mode %d.\n",
                Mode));
            break;

        default:
            break;
    }
}
```

```
/* PATCH: 951213 to allow setting of hyperplane and observer position
   along nth axis, pragmatically. */
```

void ndSet(NDenum Target, float New_Value)

```
{
    switch (Target)
    {
        case ND_HYPERPLANE_POSITION:
            Current_Context->f = New_Value;
            D(printf("ND: ndSet: Set ND_HYPERPLANE_POSITION to %1.3f.\n",
                New_Value));
            break;

        case ND_OBSERVER_POSITION:
            Current_Context->r = New_Value;
```

```
            D(printf("ND: ndSet: Set ND_OBSERVER_POSITION to %1.3f.\n",
                New_Value));
            break;

        default:
            break;
    }
}
```

void ndGetdv(NDenum Requested_State, double *Out_Value)

```
/* Return value or value array of Requested_State into Out_Value */
{
    switch (Requested_State)
    {
        case ND_RAW_DEVICE_VECTOR:
        {
            int i;
            D(printf("\nndGetdv: Returning the vector: "));
            D(Put_Vector(Current_Context->Raw_Device_Vector));

           for (i=1; i<= DIMENSION_OF(Current_Context->Raw_Device_Vector); i++)
                 Out_Value[i-1] =
                     VECTOR_ELEMENT(Current_Context->Raw_Device_Vector, i);

            break;
        }

        default:
            fprintf(stderr,
                 "ND: ndGetdv: Nothing returned, incorrect parameter\n");
            break;
    }
}
```

void ndGetfv(NDenum Requested_State, float *Out_Value)

```
/* Return value or value array of Requested_State into Out_Value */
{
    switch (Requested_State)
    {
        case ND_HYPERPLANE_POSITION:
            *Out_Value = Current_Context->f;
            break;

        case ND_OBSERVER_POSITION:
            *Out_Value = Current_Context->r;
            break;

        case ND_RAW_DEVICE_VECTOR:
        {
            int i;

           for (i=1; i<= DIMENSION_OF(Current_Context->Raw_Device_Vector); i++)
                 Out_Value[i-1] = (float)
                     VECTOR_ELEMENT(Current_Context->Raw_Device_Vector, i);
            break;
        }

        default:
            fprintf(stderr,
                 "ND: ndGetfv: Nothing returned, incorrect parameter\n");
            break;
    }
}
```

NDXContext ndXCreateContext(void)

```
{
    NDXContext New_Context;

    New_Context = (NDXContext) malloc (sizeof(__NDXContext_Struct)*1);

    New_Context->r = 2.76; /* DEFAULTS... */
    New_Context->f = 1.50;
    New_Context->World_Dimension    = 3;
    New_Context->Subspace_Dimension = 3;
```

```
    New_Context->Device_Dimension   = 2;
    New_Context->Render_Mode        = ND_RENDER;
    /*
     *New_Context = DEFAULT_CONTEXT;
     */

    New_Context->World_Projection_Stack =
        New_Matrix_Stack(DEFAULT_PROJECTION_STACK_SIZE);
    Push_Matrix(New_Context->World_Projection_Stack,
        Identity_Matrix(New_Context->World_Dimension+1));

    New_Context->World_Modelview_Stack =
        New_Matrix_Stack(DEFAULT_MODELVIEW_STACK_SIZE);
    Push_Matrix(New_Context->World_Modelview_Stack,
        Identity_Matrix(New_Context->World_Dimension+1));

    New_Context->Subspace_Projection_Stack =
        New_Matrix_Stack(DEFAULT_PROJECTION_STACK_SIZE);
    New_Context->Subspace_Modelview_Stack =
        New_Matrix_Stack(DEFAULT_MODELVIEW_STACK_SIZE);

    /* Make new context current */
    ndXMakeCurrent(New_Context);
    return New_Context;
}
```

void ndXMakeCurrent(NDXContext New_Current_Context)

```
{
    Current_Context = New_Current_Context;

    D(printf("\nndXMakeCurrent: New Context has world D = %d\n",
        Current_Context->World_Dimension));
}
```

void ndXDestroyContext(NDXContext A_Context)

```
{
    /* Free memory pointed to by A_Context */
    Free_Matrix_Stack(A_Context->World_Projection_Stack);
    Free_Matrix_Stack(A_Context->World_Modelview_Stack);
    Free_Matrix_Stack(A_Context->Subspace_Projection_Stack);
    Free_Matrix_Stack(A_Context->Subspace_Modelview_Stack);

    /* MORE ... For future revisions */
}
```

```
/*****************************************************************************
 * CONTEXTS - PRIVATE FUNCTIONS
 *****************************************************************************/
```

private

```
static void Check_Context_Exists(const char *Module)
{
    if (Current_Context==NULL) {
        Fatal_Error(Module, "No NDXContext Exists Yet");
    }
}
```

A.2.3 n-*Dimensional Matrix Transformation Control*

transformations.c

```
/*****************************************************************************
 *
 *
 *          *        n-dimensional graphics library
```

```
 *************          ------------------------------
 *************
            *           A. Ellerton
            **          Project for part of BSc(Comp.Sci) Honours
 *************          Edith Cowan University
 ************           (c) A. Ellerton 1995
 *
 *    *****             Component: transformations.c
 ***********            Description: All transformation related functions
 ***        **                      such as scale/rotate/translate etc
 *           **
  *          *
 *********************
 *********************
 *
 *

*******************************************************************************/



#include "private.h"
#include <math.h>



/*******************************************************************************
 *                              PUBLIC FUNCTIONS                               *
*******************************************************************************/
```

void ndPushMatrix(void)

```
/*------------------------------------------------------------------------------*
 * Purpose: "Push" the top of the current matrix stack down, leaving a copy of
 *          that matrix at the new top position.
 *
 * Precondition:     Matrix stack is already full
 *    Process:       Give a warning
 *    Postcondition: Stack is unchanged.
 *
 * Precondition:     not (Matrix stack is already full)
 *    Postcondition: Stack top = old stack top + 1
 *                   Stack matrix[top] = stack matrix[old stack top]
 *------------------------------------------------------------------------------
*/
{
    Matrix_Stack The_Stack = *(Current_Context->Current_Stack);

    D(printf("ndPushMatrix: ABOUT TO ENTER\n"));
    Push_Matrix(The_Stack, The_Stack->Element[The_Stack->Top] );

    D(printf("ndPushMatrix: push successful, top = %d\nStack=",
       The_Stack->Top));
    D(Put_Matrix(The_Stack->Element[The_Stack->Top]));

}
```

void ndPopMatrix(void)

```
/*------------------------------------------------------------------------------*
 * Purpose: "Pop" the top of the current matrix stack S
 *
 * Precondition:     Matrix is empty:  S(Top) = 0
 *    Process:       Give a warning
 *    Postcondition: S is unchanged.
 *
 * Precondition:     Matrix is NOT empty:  S(Top) > 0
 *    Process:       Deallocate top matrix
 *                   Decrement Top counter
 *    Postcondition: S(Top) = Old S(Top) - 1
 *------------------------------------------------------------------------------
*/
{
    Matrix_Stack The_Stack = *(Current_Context->Current_Stack);
    Matrix *Old_Top_Matrix;
```

```
    Old_Top_Matrix = Pop_Matrix(The_Stack);
    Free_Matrix(Old_Top_Matrix);
}
```

void ndLoadIdentity(void)

```
/*-------------------------------------------------------------------------*
 * Purpose: M = the top of the current matrix stack, set M = identity of
 *          dimension ????
 *-------------------------------------------------------------------------*
*/
{
/*
    Set_Top_Matrix_Stack_Element(Current_Context->Current_Stack,
                                 Identity_Matrix(N));
 */

    Matrix_Stack S = *(Current_Context->Current_Stack);
    Matrix *Current_Top_Matrix = &(S->Element[S->Top]);

    int Row, Col;
    int N = *Current_Context->Current_Space_Dimension;
                    /* ****** ESTABISH N, the DIMENSION OF THE MATRIX??  */

    Free_Matrix(Current_Top_Matrix);
    *Current_Top_Matrix = Identity_Matrix(N);
   D(printf("ND: ndLoadIdentity:\n\tSet top of stack to %D identity matrix\n"));
}
```

void ndMatrixMode(NDenum Mode)

```
/* Purpose: Redirect the current context such that any modifications to a
            matrix stack, via a call such as multMatrix() etc, will effect
            the appropriate matrix.

            Note that the "current dimension" is also redirected, as this
            is required for checking, eg ndRotate().
 */
{
    switch (Mode)
    {
        case ND_WORLD_MODELVIEW:
             Current_Context->Current_Stack =
                 &Current_Context->World_Modelview_Stack;
             Current_Context->Current_Space_Dimension =
                 &Current_Context->World_Dimension;
             DD(printf("ndMatrixMode: Changed to ND_WORLD_MODELVIEW\n"));
            DD(printf("ndMatrixMode: top of world modelview matrix stack=%d\n",
                 Current_Context->World_Modelview_Stack->Top));
             break;

        case ND_WORLD_PROJECTION:
             Current_Context->Current_Stack =
                 &Current_Context->World_Projection_Stack;
             Current_Context->Current_Space_Dimension =
                 &Current_Context->World_Dimension;
             D(printf("ndMatrixMode: Changed to ND_WORLD_PROJECTION\n"));
             break;

        case ND_SUBSPACE_MODELVIEW:
             Current_Context->Current_Stack =
                 &Current_Context->Subspace_Modelview_Stack;
             Current_Context->Current_Space_Dimension =
                 &Current_Context->Subspace_Dimension;
             D(printf("ndMatrixMode: Changed to ND_SUBSPACE_MODELVIEW\n"));
             break;

        case ND_SUBSPACE_PROJECTION:
             Current_Context->Current_Stack =
                 &Current_Context->Subspace_Projection_Stack;
             Current_Context->Current_Space_Dimension =
                 &Current_Context->Subspace_Dimension;
             D(printf("ndMatrixMode: Changed to ND_SUBSPACE_PROJECTION\n"));
             break;
```

```
        default:
            Warning("ndMatrixMode: Illegal mode: %d", Mode);
            break;
    }
}
```

void ndScale(int N, double *V)

```
{
    D(printf("IN ndScale: %dD\n", N));

    Multiply_Onto_Stack
        (*Current_Context->Current_Stack,
         Scale_Matrix(*Current_Context->Current_Space_Dimension, V));

    D(printf("ndScale: Successfully pushed scale matrix"));
}
```

void ndTranslate(int N, double *V)

```
{
    D(printf("ND: ndTranslate: %dD\n", N));

    Multiply_Onto_Stack
        (*Current_Context->Current_Stack,
         Translation_Matrix(*Current_Context->Current_Space_Dimension, V));

    D(printf("ND: ndTranslate: Successfully pushed translation matrix\n"));
}
```

void ndIdentity()

```
{
    DD(printf("IN ndIdentity\n"));

    Multiply_Onto_Stack
        (*Current_Context->Current_Stack,
         Identity_Matrix(*Current_Context->Current_Space_Dimension+1));

    DD(printf("ndIdentity: Successfully pushed matrix"));
}
```

void ndRotate(int N, int A, int B, float Angle)

```
{
    DD(printf("IN ndRotate: %d-%d plane"
            " by %1.1f degrees\n", A, B, Angle));

    if (N > *Current_Context->Current_Space_Dimension)
    {
        Warning("ndRotate: Cannot build matrix because requested size of "
                "matrix (%dD)\n        is bigger than current space (%dD).\n",
                N, *Current_Context->Current_Space_Dimension);
        return;
    }
    if (A > N || B > N || A < 0 || B < 0 || A == B)
    {
       Warning("ndRotate: Requested %dD rotation plane (%d-%d) is malformed\n",
                N, A, B);
        return;
    }

    /* Multiply rotation matrix on top of current matrix stack */
   Multiply_Onto_Stack(*Current_Context->Current_Stack,  /* onto this stack */
                    /*Rotation_Matrix(N, A, B, Angle)); /* with this matrix */

                   /* 951122 DANGER: Should poss be N not current..dimension */
                   Rotation_Matrix(*Current_Context->Current_Space_Dimension,
                                   A, B, Angle));      /* with this matrix */

    DD(printf("ndRotate: Successfully pushed rotation %dD matrix, %d-%d plane"
            " by %1.1f degrees\n", N, A, B, Angle));
}
```

void ndLoadMatrixf(int N, const float *New_Matrix)

```
{
```

```
    Warning("ndLoadMatrixf - This function has not yet been implemented\n");
}
```

void ndMultMatrixf(int N, const float *New_Matrix)

```
{
    Warning("ndMultMatrixf - This function has not yet been implemented\n");
}
```

```
/*****************************************************************************
 * PRIVATE TRANSFORMATION FUNCTIONS

*****************************************************************************/
```

Private Functions

Matrix Identity_Matrix(int N)

```
{
    int i, j;
    Matrix I;

    I = New_Matrix(N, N);
    for (i=1; i<=N; i++)
        for (j=1; j<=N; j++)
            MATRIX_ELEMENT(I, i, j) = (i==j)? 1.0 : 0.0;
    return I;
}
```

Matrix Rotation_Matrix(int N, int A, int B, double Angle)

```
{
    Matrix R;
    if (N<2)
        return NULL_MATRIX;

    R = Identity_Matrix(N+1);
    MATRIX_ELEMENT(R, A, A) = MATRIX_ELEMENT(R, B, B) = cos(TO_RADIANS(Angle));
    MATRIX_ELEMENT(R, A, B) = sin(TO_RADIANS(Angle));
    MATRIX_ELEMENT(R, B, A) = -sin(TO_RADIANS(Angle));
    return R;
}
```

Matrix Scale_Matrix(int N, double *V)

```
{
    int i, j;
    Matrix S;

    /* Create a matrix of current dimensionality */
    S = New_Matrix(N+1, N+1); /* FIX ME: SHOULD BE Current_Matrix_Dimension */

    for (i=1; i<=N+1; i++)
        for (j=1; j<=N+1; j++)
            if (i==j)
                MATRIX_ELEMENT(S, i, j) = (i==N+1)? 1.0 : V[i-1];
            else
                MATRIX_ELEMENT(S, i, j) = 0.0;
    return S;
}
```

Matrix Translation_Matrix(int N, double *V)

```
{
    int i, j;
    Matrix T;

    /* Create a matrix of current dimensionality */
    T = New_Matrix(N+1, N+1); /* FIX ME: SHOULD BE Current_Matrix_Dimension */

    for (i=1; i<=N+1; i++)
        for (j=1; j<=N+1; j++)
            if (i==j)
                MATRIX_ELEMENT(T, i, j) = 1.0;
            else if (j==N+1 && i != N+1)
                MATRIX_ELEMENT(T, i, j) = V[i-1];
            else
                MATRIX_ELEMENT(T, i, j) = 0.0;
    return T;
```

```
}
```

A.2.4 n-Dimensional Vertex Submission / Projection Control

vertex.c

```
/
*****************************************************************************
 *
 *           *         n-dimensional graphics library
 *************         ------------------------------
 *************
            *         A. Ellerton
            **        Project for part of BSc(Comp.Sci) Honours
 *************        Edith Cowan University
 ************         (c) A. Ellerton 1995
 *
 *   *****            Component: vertex.c
 ***********          Description: All vertex-related functions
 ***       **
 *          **
  *         *
 *********************
 *********************
 *
 *

*****************************************************************************/

#include <GL/nd.h>
#include "private.h"
#include <stdarg.h>             /* Variable argument lists */

static void Project(void);
```

public **void ndBegin(GLenum Mode)**

```
{
    DD(printf("\n\nndBegin(): Mode = %d\n", Mode));
    glBegin(Mode);
}
```

void ndEnd(void)

```
{
   DD(printf("ndEnd().\n"));
   glEnd();
}
```

void ndVertexNf(int N, ...)

```
{
    int Index, Elements;
    va_list Arguments;
    va_start(Arguments, N);        /* Setup Arguments Pointer */

    Check_Context_Exists("ndVertexNf");


    /* want to ignore excess components, therefore check N is less than
       currently set world dimension */

    if (N > Current_Context->World_Dimension)
        Elements = Current_Context->World_Dimension;
    else
        Elements = N;


    DD(printf("ndVertexNf: ["));
    for (Index=1; Index<=Elements; Index++)
```

```
    {
        VECTOR_ELEMENT(Current_Context->Raw_World_Vector, Index) =
            va_arg(Arguments, Element_Type);

        DD(printf("(%d) %1.1f  ",
             Index, VECTOR_ELEMENT(Current_Context->Raw_World_Vector, Index))
          );
    }

    /* Set remaining components, if any, to 0.0 (if less than the
       world dimension) or 1.0 (if = world dimension == homogeneous
       component */

    for (Index=N+1; Index <=Current_Context->World_Dimension; Index++)
    {
        VECTOR_ELEMENT(Current_Context->Raw_World_Vector, Index)=
            (Index==Current_Context->World_Dimension)? 1.0: 0.0;
        DD(printf("(%d) %1.1f  ",
             Index, VECTOR_ELEMENT(Current_Context->Raw_World_Vector, Index))
          );
    }
    DD(printf("]\n"));

    Project();
}
```

void ndVertexNd(int N, ...)

```
{
    int Index, Elements;

    va_list Arguments;
    va_start(Arguments, N);         /* Setup Arguments Pointer */

    if (N > 4 /* CHANGE */)
        Elements = 4;
    else
        Elements = N;


    for (Index=1; Index<=Elements; Index++)
        D(printf("(%d) %1.1f  ",
             Index, va_arg(Arguments, double)));

    va_end(Arguments);              /* 'clean up' arguments */
    D(printf("]\n"));
}
```

void ndNormalNfv(int N, float *V)

```
{
    Warning("ndNormalNFv: Normals not yet implemented\n");
}
```

void ndVertexNfv(int N, float *V)

```
{
    int Index, Elements;

    Check_Context_Exists("ndVertexNfv");


    /* want to ignore excess components, therefore check N is less than
       currently set world dimension */

    if (N > Current_Context->World_Dimension)
        Elements = Current_Context->World_Dimension;
    else
        Elements = N;


    D(printf("ndVertexNfv: ["));
    for (Index=1; Index<=Elements; Index++)
    {
```

```
        VECTOR_ELEMENT(Current_Context->Raw_World_Vector, Index) =
           (Element_Type) V[Index-1];
        D(printf("%1.1f  ", VECTOR_ELEMENT(Current_Context->Raw_World_Vector,
                                Index)
                )
         );
    }

    /* Set remaining components, if any, to 0.0 (if less than or equal to the
       world dimension) or 1.0 (if == homogeneous component */

    for (Index=N+1; Index <=Current_Context->World_Dimension+1; Index++)
    {
        VECTOR_ELEMENT(Current_Context->Raw_World_Vector, Index)=
            (Index==Current_Context->World_Dimension+1)? 1.0: 0.0;
        D(printf("%1.1f  ", VECTOR_ELEMENT(Current_Context->Raw_World_Vector,
                                Index)
                )
         );
    }
    D(printf("]\n"));

    Project();
}
```

void ndVertexNdv(int N, double *V)

```
{
    Warning("ndVertexNdv: This function is not yet implemented\n");
}
```

Private Functions

static Vector Project_Vector_N_To_K_Dimensions(Vector V, int N, int K)

```
/*-------------------------------------------------------------------------
 * Purpose: To project vector V from N-dimensions to K-dimensions.
 *------------------------------------------------------------------------- */
{
    /*double r = 2.76,f=1.5, q;*/
    double r,f,q;
    int i, d;
    Vector R;                       /* = New_Vector(DIMENSION_OF(V));*/

    D(printf("\nND: Project_Vector_N_To_K_Dimensions()\n"));

    /* ------------------------------------------------------------------
       REPEATED PROJECTION
       -------------------
       Start at dimension N, project to dimension K, Step down
       by 1 dimension each iteration.

      "d" loop counter = dimension to project FROM, therefore only goes to one
          ABOVE the K, ie d > K
       ------------------------------------------------------------------*/

    R = Copy_Vector(V);
    r = Current_Context->r;
    f = Current_Context->f;

   D(printf("\tProjection loop, from %dD to %dD begun (r=% 1.3f, f=% 1.3f).\n",
       N, K, r, f));
    for (d=N; d>K; d--)
    {
        /* Derive q from so-far projected vector */
        q = (r-f)/(r-VECTOR_ELEMENT(R,d));

        D(printf("\t\tProject from %dD to %dD: [ ", d, d-1));
        /* Populate all salient members of R (all but last) with R = qR */
        for (i=1; i<d; i++)
        {
            VECTOR_ELEMENT(R, i) = q * VECTOR_ELEMENT(R, i);
            D(printf(" %1.1f", VECTOR_ELEMENT(R, i)));
        }
        D(printf("] \n"));
    }
    D(printf("\tProjection loop, from %dD to %dD complete.\n", N, K));
```

```
    /* Debuggin only Feedback */

    D(printf("\tProjection Result: %dD vector = ", K));
    D(Put_Vector(R));

    /* Augment size of device vector to reflect projection */
    DIMENSION_OF(R) = K;
    return R;
}
```

static void Project(void)

```
{
    Matrix_Stack Stack;

    D(printf("\nND:Project()\n\tMatrix to transform world vector = \n"));
    D(Put_Matrix(TOP_MATRIX(Current_Context->World_Modelview_Stack)));

    Free_Vector(Current_Context->Transformed_World_Vector);

    /*-------------------------------------------------------------------------
       Transformed World Vector = World_Modelview_Matrix * Raw_World_Vector
      -------------------------------------------------------------------------*/

    Current_Context->Transformed_World_Vector =
        Multiply_Matrix_By_Vector
            (TOP_MATRIX(Current_Context->World_Modelview_Stack),
             Current_Context->Raw_World_Vector);

    D(printf("Project(): Transformed world vector = "));
    D(Put_Vector(Current_Context->Transformed_World_Vector));

    if (Current_Context->Subspace_Projection_Hint == ND_TRUE)
    {
       /* Project to a subspace first, then project to device */
        /*Warning("Project(): Subspace projection not yet implemented\n");*/


        /* 1. Project from world to subspace */
        Free_Vector(Current_Context->Raw_Subspace_Vector);

        Current_Context->Raw_Subspace_Vector =
            Project_Vector_N_To_K_Dimensions
                (Current_Context->Transformed_World_Vector,
                Current_Context->World_Dimension,          /* FROM DIMENSION */
                Current_Context->Subspace_Dimension);        /* TO DIMENSION */

        (printf("Project(): raw subspace vector = "));
        (Put_Vector(Current_Context->Raw_Subspace_Vector));



        /* 2. Transform within subspace */
        Free_Vector(Current_Context->Transformed_Subspace_Vector);

        Current_Context->Transformed_Subspace_Vector =
            Multiply_Matrix_By_Vector
                (TOP_MATRIX(Current_Context->Subspace_Modelview_Stack),
                 Current_Context->Raw_Subspace_Vector);

        (printf("Project(): Transformed subspace vector = "));
        (Put_Vector(Current_Context->Transformed_Subspace_Vector));


        /* 3. Project from subspace to device */
        Free_Vector(Current_Context->Raw_Device_Vector);

        Current_Context->Raw_Device_Vector =
            Project_Vector_N_To_K_Dimensions
                (Current_Context->Transformed_Subspace_Vector,
                Current_Context->Subspace_Dimension,       /* FROM DIMENSION */
                Current_Context->Device_Dimension);          /* TO DIMENSION */
    }
    else
    {
        Free_Vector(Current_Context->Raw_Device_Vector);
```

```
        /*-------------------------------------------------------------------
          Project from world dimension to device dimension
          -------------------------------------------------------------------*/

        Current_Context->Raw_Device_Vector =
            Project_Vector_N_To_K_Dimensions
                (Current_Context->Transformed_World_Vector,
                Current_Context->World_Dimension,          /* FROM DIMENSION */
                Current_Context->Device_Dimension);          /* TO DIMENSION */
    }



    /* Debug Feedback */
    D(printf("ND:Project():\n\tIn: %dD -> GL Out: %dD\n",
        Current_Context->World_Dimension,
        Current_Context->Device_Dimension));

    D(printf("\t%dD world vector = ",
        Current_Context->World_Dimension));
    D(Put_Vector(Current_Context->Raw_World_Vector));

    D(printf("\t%dD transformed world vector = ",
        Current_Context->World_Dimension));
    D(Put_Vector(Current_Context->Transformed_World_Vector));

    D(printf("\t%dD device vector = ",
        Current_Context->Device_Dimension));
    D(Put_Vector(Current_Context->Raw_Device_Vector));

    /* Now draw using OpenGL, only if in ND_RENDER mode */
    if (Current_Context->Render_Mode == ND_RENDER)
        Current_Context->OpenGL_Draw_Function
            (ACCESS_ELEMENTS(Current_Context->Raw_Device_Vector));
    else
        D(printf("Not rendering because currently in ND_RENDER mode\n"));
}
```

A.3 Underlying Architecture Source Code

A.3.1 Private Function Interface

The header file private.h contains all type and function specifications for the underlying, or "private" functionality of the ND library. That is, all types and functions that are not directly available to a user of the ND library, are listed within here, for access by the principal architecure, as listed above.

nd/lib/nd/private.h

```
/*****************************************************************************
 *
 *
 *           *         n-dimensional graphics library
 ************           ------------------------------
 ************
            *          A. Ellerton
            **         Project for part of BSc(Comp.Sci) Honours
 ************         Edith Cowan University
 ************         (c) A. Ellerton 1995
 *
```

```
 *   *****              Component: ndprivate.h
 ***********            Description: private types/functions for nd
 ***       **
 *          **
  *         *
 *********************
 *********************
 *
 *

*******************************************************************************/

#ifndef ND_PRIVATE_H
#define ND_PRIVATE_H

#include <GL/nd.h>
#include <GL/glx.h>  /* For GLX context */

#ifdef __cplusplus
extern "C" {
#endif


/* ---------------------------------------- *
 * ADD ALL PUBLIC DEFINITIONS AFTER THIS LINE *
 * ---------------------------------------- */
```

Debugging Control

```
#define DEBUG NOT

#if (DEBUG == 2)
#   define D(Anything) Anything
#   define DD(Anything) Anything
#   define UP_TO_HERE D(printf("Up to line %d of %s.\n", __LINE__, __FILE__));
#   error DEBUG LEVEL SET AT 2

#elif (DEBUG == 1)
#   define D(Anything) Anything
#   define DD(Anything)
#   define UP_TO_HERE D(printf("Up to line %d of %s.\n", __LINE__, __FILE__));

#else
#   define D(Anything)
#   define DD(Anything)

#endif

#   include <stdio.h>                /* printf, if req'd */

/
*******************************************************************************
 * MANAGER/UTILITY FUNCTIONS

*******************************************************************************
/
#define TO_RADIANS(angle) ((angle)*M_PI/180.0)

typedef double Element_Type;
```

vector data-structure

```
typedef struct
{
    int Elements;
    Element_Type *Element;
}   Vector;
```

matrix data-structure

```
typedef struct
{
    int Rows, Columns;
    Element_Type **Element;
}   Matrix;
```

matrix stack data-structure

```
/* Matrix Stack Operation.
 * A given stack S has a maximum of m and a minimum of 1 elements.
 * Each element s(i) is a n-dimensional matrix, each element does not
```

```
 * necessarily have to be the same size- therefore an element is a pointer
 * to a n-dimensional matrix. Therefore an array of elements is an array
 * of pointers to matrices.
 *-------------------------------------------------------------------------
*/
typedef struct
{
    int Top;                          /* Current size of stack */
    int Size;                        /* Maximum number of elements in stack */
    Matrix  *Element;                 /* An array of matrices */
}   Matrix_Stack_Struct, *Matrix_Stack;
```

rendering context structure

```
typedef struct __NDXContext_Struct
{
    /*-----------------
      STACKS
      ----------------- */
    Matrix_Stack  World_Projection_Stack;
    Matrix_Stack  World_Modelview_Stack;
    Matrix_Stack  Subspace_Projection_Stack;
    Matrix_Stack  Subspace_Modelview_Stack;
    Matrix_Stack *Current_Stack;      /* Points to current stack */

    /*-----------------
      EXTERNAL
      ----------------- */
    GLXContext    OpenGL_Context;

    /*-----------------
      DIMENSIONS
      ----------------- */
    int World_Dimension;
    int Subspace_Dimension;
    int Device_Dimension;             /* 2, 3 or 4->determines GL func to use*/
    int *Current_Space_Dimension;


    /*-----------------
      HINTS/PREFERENCES
      ----------------- */

    NDenum Use_GL_Hint;               /* When to use GL */
    NDenum Subspace_Projection_Hint; /* True / False */
    NDenum Render_Mode;



    /*--------------------------
      VECTORS (USEFUL FOR GET())
      -------------------------- */

    Vector Raw_World_Vector;              /* Input vector, before any
                                             manipulation*/
    Vector Transformed_World_Vector;      /* Input world coordinate vector,
                                             after transformation, but not
                                             projection */
    Vector Raw_Subspace_Vector;           /* World vector after transformation
                                             and projection to subspace, (if
                                             subspace proj is TRUE) */

    Vector Transformed_Subspace_Vector;  /* Subspace vector after
                                             transformation in subspace*/
    Vector Raw_Device_Vector;             /* Subspace vector after
                                             transformation and projection to
                                             device. NB device may be 3D, like

    /*--------------------------
      OPEN GL STUFF...
      -------------------------- */
    void (*OpenGL_Draw_Function)(const GLdouble *V);

    /*--------------------------
      PATCH
      -------------------------- */

    float r, f;          /* observer/plane position along nth axis.
                            Added 951213 ... _should_ be part of projection
```

```
                                transformations... */

} __NDXContext_Struct;


/
*****************************************************************************
 * GLOBAL VARIABLES

*****************************************************************************
/

extern NDXContext Current_Context;
extern const __NDXContext_Struct DEFAULT_CONTEXT;


/
*****************************************************************************
 * VECTOR DEFINITIONS

*****************************************************************************
/
extern const Vector NULL_VECTOR;
#define ACCESS_ELEMENTS(V)  ((V).Element)
#define VECTOR_ELEMENT(V,INDEX)   ((V).Element[ (INDEX)-1 ])
#define DIMENSION_OF(V)   ((V).Elements)
#define IS_NULL_VECTOR(V)  ((V).Elements==NULL_VECTOR.Elements && \
                            (V).Element == NULL_VECTOR.Element)
```

extern Vector Copy_Vector(Vector);

```
/*-----------------------------------------------------------------------*
 * Purpose: Copy vector and return a pointer to the new (dynamic) vector
 *-----------------------------------------------------------------------
*/
```

extern void New_Vector(int Number_Of_Elements, Vector *New_Vector);

```
/*-----------------------------------------------------------------------*
 * Purpose: Create a vector of size Elements.
 *-----------------------------------------------------------------------
*/
```

extern void Free_Vector(Vector A_Vector);

```
/*-----------------------------------------------------------------------*
 * Purpose: Free the memory stored for A_Vector
 *
 *-----------------------------------------------------------------------
*/
```

extern Vector Set_Vector(int Number_Of_Elements, ...);

```
/*-----------------------------------------------------------------------*
 * Purpose: Create and populate a vector (size = Elements) and populate with
 *          using contents of variable length argument list - must have Elements
 *           floats in it.
 *-----------------------------------------------------------------------
*/
```

extern void Put_Column_Vector(Vector V);

```
/*-----------------------------------------------------------------------*
 * Purpose: Put an ascii representation of M to stdout
 *          Displays in form of a COLUMN vector. See also: Put_Vector()
 *-----------------------------------------------------------------------
*/
```

extern void Put_Vector(Vector V);

```
/*-----------------------------------------------------------------------*
 * Purpose: Put an ascii representation of M to stdout
 *          Displays in form of a ROW vector. See also: Put_Column_Vector()
 *-----------------------------------------------------------------------
*/


/
*****************************************************************************
```

```
 * MATRIX DEFINITIONS

*****************************************************************************
/
extern const Matrix NULL_MATRIX;

#define MATRIX_ELEMENT(M,ROW,COL)   ((M).Element[ (ROW)-1 ][ (COL)-1] )
#define COLUMNS_OF(M)   ((M).Columns)
#define ROWS_OF(M)   ((M).Rows)
#define IS_NULL_MATRIX(M)  ((M).Element == NULL || *(M).Element == NULL)
extern int Dimension(Matrix A_Matrix);

extern Matrix Copy_Matrix(Matrix A_Matrix);
/*---------------------------------------------------------------------------*
 * Purpose: Copy A_Matrix and return a pointer to the new (dynamically
 *          allocated) matrix
 *
 *---------------------------------------------------------------------------
*/

extern Matrix New_Matrix(int Rows, int Cols);
/*---------------------------------------------------------------------------*
 * Purpose: Create a Rows x Cols array of matrix elements and return the
 *          pointer to the new (dynamically allocated) matrix
 *
 *---------------------------------------------------------------------------
*/

extern NDboolean Multiply_Matrices(Matrix *R, Matrix A, Matrix B);
/*---------------------------------------------------------------------------*
 * Purpose: Calculate and return result of A * B
 *---------------------------------------------------------------------------
*/

extern Vector Multiply_Matrix_By_Vector(Matrix M, Vector V);
/*---------------------------------------------------------------------------*
 * Purpose: Calculate and return result of M.V
 *          Note that this is PRE-multiplication, and M.V != V.M generally
 *---------------------------------------------------------------------------
*/

extern void Free_Matrix(Matrix *A_Matrix);
/*---------------------------------------------------------------------------*
 * Purpose: Free the memory stored for A_Matrix
 *
 *---------------------------------------------------------------------------
*/

extern Matrix Set_Matrix(int Rows, int Cols, ...);
/*---------------------------------------------------------------------------*
 * Purpose: Create and populate A_Matrix to be Rows x Cols in size, using
 *          variable length argument list - which must have exactly Row*Cols
 *          floats in it - to populate the values.
 *---------------------------------------------------------------------------
*/

extern void Put_Matrix(Matrix M);
/*---------------------------------------------------------------------------*
 * Purpose: Put an ascii representation of M to stdout
 *
 *---------------------------------------------------------------------------
*/

/
*****************************************************************************
 * MATRIX STACK DEFINITIONS

*****************************************************************************
/

#define EMPTY_STACK -1
#define STACK_IS_EMPTY(S)  ((S)->Top == EMPTY_STACK)
#define STACK_IS_FULL(S)   ((S)->Top == (S)->Size-1)
#define TOP_MATRIX(S)      ((S)->Element[(S)->Top])
```

```
extern void       Free_Matrix_Stack(Matrix_Stack The_Stack);
/*-------------------------------------------------------------------------*
 * Purpose:
 *
 *-------------------------------------------------------------------------
*/

extern void       Initialise_Stack(Matrix_Stack *The_Stack, int New_Size);
/*-------------------------------------------------------------------------*
 * Purpose:
 *
 *-------------------------------------------------------------------------
*/

extern void Multiply_Onto_Stack(Matrix_Stack S, Matrix M);
/*-------------------------------------------------------------------------*
 * Purpose: To multiply matrix M by the top stack of matrix S (if it exists)
   Precondition: S is empty
      Process:   Copy M to top of new stack
      Postcondition: M sole matrix on stack S

   Precondition: S is NOT empty
      Process:   Replace S's top matrix T with T.M
      Postcondition: S.Top_Matrix = S.Top_Matrix * M
 *-------------------------------------------------------------------------
*/

extern Matrix_Stack New_Matrix_Stack(int New_Size);

extern Matrix *   Pop_Matrix(Matrix_Stack S);

extern void       Push_Matrix(Matrix_Stack S, Matrix M);

extern void       Put_Matrix_Stack(Matrix_Stack S);
/*-------------------------------------------------------------------------*
 * Purpose: Put an ascii representation of S to stdout
 *
 *-------------------------------------------------------------------------
*/

/
*****************************************************************************
 * TRANSFORMATION MATRIX FUNCTIONS

*****************************************************************************
/

Matrix Identity_Matrix(int N);
/*-------------------------------------------------------------------------*
 * Purpose:  Create a n-dimensional NON-homogeneous identity matrix
 *-------------------------------------------------------------------------
*/

Matrix Rotation_Matrix(int N, int A, int B, double Angle);
/*-------------------------------------------------------------------------*
 * Purpose:  Produce a N x N Homogeneous matrix (i.e. N+1 x N+1 sized) that will
 *           effect a rotation in the A-B plane by Angle degrees.
 *
 * Precondition: N >= 2
 * Process:  Create R such that R[i][j] = 1 : i equal to j
 *                                        0 : i not equal to j
 *           EXCEPT
 *                              R[a][a] = R[b][b] = cos(angle)
 *                              R[a][b] = -R[b][a] = sin(angle)
 *           return R;
 * Postcondition:
 *
 * Precondition: N < 2
 * Process:      return NULL;
 * Postcondition:
 *
```

```
 * Reference: Noll (1967) A Computer Technique for Displaying n-dimensional
 *            hyperobjects. Comm of the ACM 10(8)
 *-----------------------------------------------------------------------
*/
```

Matrix Scale_Matrix(int N, double *V);

```
/*-----------------------------------------------------------------------*
 * Purpose: Produce a N-dimensional homogeneous matrix (N+1 x N+1 size)
 *          to effect a scaling in each dimension, as described by vector V.
 *-----------------------------------------------------------------------
*/
```

Matrix Translation_Matrix(int N, double *V);

```
/*-----------------------------------------------------------------------*
 * Purpose: Produce a N-dimensional homogeneous matrix (N+1 x N+1 size)
 *          to effect a translation in each dimension, as described by vector V.
 *-----------------------------------------------------------------------
*/


/
*************************************************************************
 * DIAGNOSTIC/UTILITY FUNCTIONS

*************************************************************************
/
#define Verify_Allocation(Item,Failure_Message) \
   if (Item == NULL) { fprintf(stdout, "Out of Memory: %s\n", Failure_Message);
fprintf(stderr, "Out of Memory: %s\n", Failure_Message); }
```

extern void Warning(const char *Format, ...);
extern float Random_Float(float Minimum, float Maximum);
extern int Random_Int(int Minimum, int Maximum);

extern void Non_Fatal_Error(const char *Module, const char *Message);

```
/*-----------------------------------------------------------------------*
 * Purpose: Display message to stderr, form:
 *
 * ND Non-Fatal Error: Module=>"Module": Message
 *-----------------------------------------------------------------------
*/
```

extern void Fatal_Error(const char *Module, const char *Message);

```
/*-----------------------------------------------------------------------*
 * Purpose: Display message to stderr, then exit with non-zero code.
 *          Form of the display:
 *
 * ND FATAL ERROR: Module=>"Module": Message
 *-----------------------------------------------------------------------
*/
```

extern void Check_Context_Exists(const char *Module);

```
/*-----------------------------------------------------------------------*
 * Purpose: Checks if a context is currently associated. If not, exits.
 *-----------------------------------------------------------------------
*/

/* ADD NOTHING AFTER THIS LINE */
#ifdef __cplusplus
}
#endif

#endif
```

A.3.2 Matrix Control

matrix.c

```
/
*****************************************************************************
 *
 *
 *           *          n-dimensional graphics library
 *************          ------------------------------
 *************
            *          A. Ellerton
            **         Project for part of BSc(Comp.Sci) Honours
 *************         Edith Cowan University
 ************          (c) A. Ellerton 1995
 *
 *   *****             Component: matrix.c
 ***********           Description: Strictly matrix-related functions
 ***       **
 *          **
  *         *
 *********************
 *********************
 *
 *

*****************************************************************************/

#include <GL/nd.h>
#include "private.h"
#include <stdarg.h>

/*****************************************************************************
 * TYPES/DEFINITONS ETC

*****************************************************************************/

const Matrix NULL_MATRIX = {0, 0, NULL};


/*****************************************************************************
 * MATRICES - AVAILABLE FUNCTIONS

*****************************************************************************/
```

Matrix Copy_Matrix(Matrix The_Original)

```
/* Return matrix The_Copy, a copy of Original
 *
 * 951108 I have duplicated the code in New_Matrix in the hope it will increase
   efficiency b/c Copy_Matrix is used frequently.
 * 951108.2
 */
{
    int Row;
    Matrix The_Copy;
    The_Copy = New_Matrix(ROWS_OF(The_Original), COLUMNS_OF(The_Original));
    DD(printf("Copy_Matrix: %d x %d matrix\n", The_Original.Rows,
The_Original.Columns));
    /* Now copy in all elements ... */

    /* Setup values */
    The_Copy.Rows = The_Original.Rows;
    The_Copy.Columns = The_Original.Columns;
    The_Copy.Element =
        (Element_Type **) malloc (sizeof(Element_Type *) * The_Original.Rows);
    if (The_Copy.Element == NULL) /* out of memory */
    {
        Warning("Out of memory when creating a new matrix\n");
        return NULL_MATRIX;
    }
    /* Allocate each row */
    for (Row=0; Row<The_Original.Rows; Row++)
    {
```

```
        The_Copy.Element[Row] =
        (Element_Type *) calloc (sizeof(Element_Type), The_Original.Columns);
        if (The_Copy.Element[Row] == NULL) /* out of memory */
        {
            Warning("Out of memory when creating a row in a matrix\n");
            return NULL_MATRIX;
        }

        /* Copy each element */
        memcpy(The_Copy.Element[Row], The_Original.Element[Row],
               The_Original.Columns * sizeof(Element_Type));
        DD(printf("COPY_MATRIX: Row %d -> %p\n", Row, The_Copy.Element[Row]));
    }
    return The_Copy;
}
```

Matrix New_Matrix(int Number_Of_Rows, int Number_Of_Columns)

```
{
    int Row;
    Matrix M;

    /* Setup values */
    M.Rows = Number_Of_Rows;
    M.Columns = Number_Of_Columns;
    M.Element =
        (Element_Type **) malloc (sizeof(Element_Type *) * Number_Of_Rows);
    if (M.Element == NULL) /* out of memory */
    {
        Warning("Out of memory when creating a new matrix\n");
        return NULL_MATRIX;
    }
    /* Allocate each row */
    for (Row=0; Row<Number_Of_Rows; Row++)
    {
        M.Element[Row] =
        (Element_Type *) calloc (sizeof(Element_Type), Number_Of_Columns);
        Verify_Allocation(M.Element[Row], "New Matrix");
        if (M.Element[Row] == NULL) /* out of memory */
        {
            Warning("Out of memory when creating a row in a matrix\n");
            return NULL_MATRIX;
        }
        DD(printf("NEW_MATRIX: Row %d -> %p\n", Row, M.Element[Row]));
    }
    return M;
}
```

void Free_Matrix(Matrix *M)

```
/* deallocate dynamic memory in matrix M */
{
    int Row;

    if (IS_NULL_MATRIX(*M))
    {
        DD(printf("Free_Matrix(): Matrix is already NULL\n"));
        return;
    }

    /* Deallocate each row */
    for (Row=0; Row < M->Rows; Row++)
    {
        if (M->Element[Row] != NULL)
        {
            free (M->Element[Row]);
            M->Element[Row] = NULL;
        }
        else
          fprintf(stderr, "WARNING: Free_Matrix: Not freeing row %d in a matrix
- "
                          "Pointer is already null?\n", Row+1);
    }

    if (M->Element != NULL)
    {
        DD(printf("Freeing matrix...\n"));
```

```
        free (M->Element);
        M->Element = NULL;
    }
    else
        fprintf(stderr, "WARNING: Free_Matrix: Not freeing matrix - "
                        "Pointer already null?\n");
}
```

Matrix Set_Matrix(int Rows, int Cols, ...)

```
{
    int Row, Col;
    Matrix M;
    Element_Type Value;

    va_list Arguments;
    va_start(Arguments, Cols);        /* Setup Arguments Pointer */

    M = New_Matrix(Rows,Cols);
    for (Row=1; Row<=Rows; Row++)
        for (Col=1; Col<=Cols; Col++)
           MATRIX_ELEMENT(M,Row,Col) = va_arg(Arguments, Element_Type);
    va_end(Arguments);            /* 'clean up' arguments */
    return M;
}
```

void Put_Matrix(Matrix M)

```
{
    int Row, Col;
    DD(printf("   Put_Matrix: %d x %d matrix\n", M.Rows, M.Columns));
    for (Row=1; Row<=M.Rows; Row++)
    {
        printf("  %c ", (Row==1)? '(' : ' ');
        printf("  %p ", M.Element[Row-1]);
        for (Col=1; Col<=M.Columns; Col++)
            printf("% 3.3f ", MATRIX_ELEMENT(M, Row, Col));
        printf("%c\n", (Row==ROWS_OF(M))? ')':' ');
    }
}
```

int Dimension(Matrix A_Matrix)

```
/* return the dimensions of a matrix, if square, 0 if not square */
{
    return (A_Matrix.Rows==A_Matrix.Columns)? A_Matrix.Rows : 0;
}
```

NDboolean Multiply_Matrices(Matrix *R, Matrix A, Matrix B)

```
/*****************************************************************************
 * Purpose: Calculate and return result of A * B
 * References: Foley et al (1991) p. 1103 or most vector/matrix texts.
 *
 * Precondition: A is an n x k matrix and
 *               B is an k x p matrix and
 *               n, k, p != 0
 *
 *   Process: Allocate R, the result matrix, as a n * p matrix
 *
 *                            k
 *           R[row][col] = SUM  a[row,s) * b[s][col]
 *                           s=1
 *
 *                         : 1 <= row <= n, 1 <= col <= p
 *
 *           For a 4x4 matrix multiplied by another 4x4 matrix this is:
 *                       = a[row][1] * b[1][col] +
 *                         a[row][2] * b[2][col] +
 *                         a[row][3] * b[3][col] +
 *                         a[row][4] * b[4][col]
 *           Return R
 *    Postcondition: A and B are unchanged
 *
 * Precondition: not (A is an n x k matrix and
 *                    B is an k x p matrix and
 *                    n, k, p != 0)
 *    Process: return NULL_MATRIX
```

```
 *    Postcondition: A and B are unchanged
 ******************************************************************************/
{
    Element_Type Value;
    int s, n, k, p, Row, Column;

    n = ROWS_OF(A);
    p = COLUMNS_OF(B);
    k = COLUMNS_OF(A);


    if (n==0 || p == 0 || k == 0 || COLUMNS_OF(A) != ROWS_OF(B))
    {
        /* the matrices A and B cannot be multiplied */
        Warning("Matrices dimensions are 0 or not compatible\n");
        *R = NULL_MATRIX;
        return ND_FALSE;
    }

    Free_Matrix(R);
    *R = New_Matrix(n, p);

    for (Row=1; Row<=k; Row++)
        for (Column=1; Column<=k; Column++)
        {
            Value = 0.0;
            for (s=1; s<=k; s++)
                Value += MATRIX_ELEMENT(A,Row,s) * MATRIX_ELEMENT(B,s,Column);
            MATRIX_ELEMENT(*R,Row,Column) = Value;
        }
    return ND_TRUE;
}
```

Vector Multiply_Matrix_By_Vector(Matrix M, Vector V)

```
/*-----------------------------------------------------------------------------
 * Purpose: Calculate and return result of M.V
 *          Note that this is PRE-multiplication, and M.V != V.M generally
 *
 * Precondition: V has N elements, either 1 x N (row vector) or N x 1 (column)
 *               where N > 0, and
 *               M is at least N x N matrix
 *
 *   Process: Allocate R, the result vector, as a N element Vector
 *
 *                          N
 *            R[element] = SUM  M[element,s] * V[s]
 *                         s=1
 *
 *                        : 1 <= element <= n
 *
 *            For a 4x4 matrix multiplied by 4D vector this is:
 *            R[element]   = M[element][1] * V[1] +
 *                           M[element][2] * V[2] +
 *                           M[element][3] * V[3] +
 *                           M[element][4] * V[4]
 *            Return R
 *
 *    Postcondition: M and V are unchanged
 *
 * Precondition: not (N >0 and M is at least N x N matrix)
 *
 *   Process: Display a warning, N must be > 0 and M must be AT LEAST N x N
 *            return NULL_VECTOR;
 *
 *   Postcondition: M and V are unchanged
 *
 * Assumptions: Assume, for efficiency at run-time, that R is allocated to
 *              an appropriate size.
 *---------------------------------------------------------------------------
*/
{
    int Element, s;
    int N = DIMENSION_OF(V);
    Element_Type Value;
    Vector R;
```

```
    D(printf("ND: Multiply_Matrix_By_Vector(): \n\tMatrix = \n"));
    D(Put_Matrix(M));
    D(printf("\tVector is %dD, and = ", N));
    D(Put_Vector(V));

    if (N <= 0)
    {
         Warning("ND: Multiply_Matrix_By_Vector(): \n\tVector size < 0 - "
                 "NO MULTIPLICATION PERFORMED\n");
         return V;
    }

    if (ROWS_OF(M) != COLUMNS_OF(M) ||
        ROWS_OF(M) == 0 || COLUMNS_OF(M) == 0)
    {
         Warning("ND: Multiply_Matrix_By_Vector(): \n\t"
            "Matrix is not square or is null-RESULT SET TO ORIGINAL VECTOR\n");
         return V;
    }

    if (N > ROWS_OF(M))
    {
         Warning("ND: Multiply_Matrix_By_Vector(): \n\t"
              "Vector is bigger than matrix - NO MULTIPLICATION PERFORMED\n");
         return V;
    }

    New_Vector(N, &R);


    for (Element=1; Element<=N; Element++)
    {
        /* DD(printf("R[%d] = ", Element)); */
        Value = 0.0;
        for (s=1; s<=N; s++)
        {
             Value += MATRIX_ELEMENT(M,Element,s) * VECTOR_ELEMENT(V,s);
             /* DD(printf("+ %1.3f * %1.3f  ",MATRIX_ELEMENT(M,Element,s),
VECTOR_ELEMENT(V,s))); */
        }
        VECTOR_ELEMENT(R,Element) = Value;
        /* DD(printf("\n")); */
    }
    D(printf("ND: Multiply_Matrix_By_Vector(): \n\tRESULT = "));
    D(Put_Vector(R));
    return R;
}
```

A.3.3 Matrix Stack Control

matrix_stack.c

```
/
*****************************************************************************
 *
 *
 *           *          n-dimensional graphics library
 *************          ------------------------------
 *************
            *          A. Ellerton
            **         Project for part of BSc(Comp.Sci) Honours
 *************         Edith Cowan University
 ************          (c) A. Ellerton 1995
 *
 *   *****             Component: matrix_stack.c
 **********            Description: All matrix stack functions
 ***       **
 *          **
  *         *
 ********************
```

```
 *********************
 *
 *

*****************************************************************************/

#include <GL/nd.h>
#include "private.h"
#include <stdarg.h>


/*****************************************************************************
 * MATRIX STACKS - AVAILABLE FUNCTIONS

*****************************************************************************/
```

Matrix_Stack New_Matrix_Stack(int New_Size)

```
{
    Matrix_Stack S;

    S = (Matrix_Stack) malloc (sizeof(Matrix_Stack_Struct));
    S->Top = EMPTY_STACK;
    S->Size = New_Size;

    /* Allocate an array of elements */
    S->Element = (Matrix *) malloc (sizeof(Matrix)*New_Size);
    Verify_Allocation(S->Element, "Building a new stack");

   D(printf("Allocated a new stack capable of holding %d matrices\n",S->Size));
    return S;
}
```

void Free_Matrix_Stack(Matrix_Stack S)

```
{
    int Index;
    D(printf("want to free a matrix stack\n"));
    if (STACK_IS_EMPTY(S))
        Non_Fatal_Error("Free_Matrix_Stack", "cannot dealloc an empty stack");
    else
    {
        for (Index=0; Index <= S->Top; Index++)
            /* 951119 WAS: Free_Matrix(S->Element[Index]); */
            Free_Matrix(&(S->Element[Index]));

        /* Tell the stack its empty */
        S->Top = EMPTY_STACK;
        D(printf("Deallocated %d element(s) of a matrix stack\n", Index));
    }
}
```

void Put_Matrix_Stack(Matrix_Stack S)

```
{
    int Index;
    if (STACK_IS_EMPTY(S))
        printf("\n  Stack Matrix is empty. \n", Index+1);

    for (Index=0; Index <= S->Top; Index++)
    {
        printf("\n  Stack Matrix [%d]: \n", Index+1);
        Put_Matrix(S->Element[Index]);
    }
    printf("\n\n");
}
```

void Multiply_Onto_Stack(Matrix_Stack S, Matrix M)

```
{
    if (STACK_IS_EMPTY(S))
    {
        Non_Fatal_Error("Multiply_Onto_Stack",
                        "Stack is empty - adding matrix as first element\n");
        S->Top++;
        S->Element[S->Top] = Copy_Matrix(M);
        DD(printf("Multiply_Onto_Stack: successful\n"));
    }
```

```
    else
    {
        Matrix R = NULL_MATRIX;
        DD(printf("Multiply_Onto_Stack():\n   Matrix Stack Top: %d x %d\n"
                "   XForm Matrix:     %d x %d\n",
                ROWS_OF(S->Element[S->Top]), COLUMNS_OF(S->Element[S->Top]),
                ROWS_OF(M), COLUMNS_OF(M)));
        D(printf("ND: Multiply_Onto_Stack:\n\tTransform matrix = \n"));
        D(Put_Matrix(M));
        Multiply_Matrices(&R, S->Element[S->Top], M); /* R = T.M */

        if (IS_NULL_MATRIX(R))
        {
            Non_Fatal_Error("Multiply_Onto_Stack",
                 "Matrix multiplication failure!\n");
            return;
        }

        /* if stack is empty, put new matrix at first (0) position.
           Otherwise put it at the next */

        S->Element[S->Top] = Copy_Matrix(R);
        D(printf("ND: Multiply_Onto_Stack: successful. New matrix:\n"));
        D(Put_Matrix(R));
        Free_Matrix(&R);
    }
}
```

void Push_Matrix(Matrix_Stack S, Matrix M)

```
{
    if (STACK_IS_FULL(S))
    {
        Non_Fatal_Error("PushMatrix",
                        "Stack is at maximum size - cannot push matrix\n");
    }
    else
    {
        D(printf("\nND: Push_Matrix: \n\tNew Matrix: \n"));
        D(Put_Matrix(M));

        /* if stack is empty, put new matrix at first (0) position.
           Otherwise put it at the next */

        S->Top++;
        S->Element[S->Top] = Copy_Matrix(M);
        D(printf("\nND: Push_Matrix: \n\t"
            "successful - Newly pushed matrix[Top=>%d] = \n", S->Top));
        D(Put_Matrix(  S->Element[S->Top]  ));
        D(printf("\nND: Push_Matrix: PREVIOUSLY TOP matrix[Old Top=>%d] = \n",
             S->Top-1));
        D(Put_Matrix(  S->Element[S->Top-1]  ));

    }
}
```

Matrix * Pop_Matrix(Matrix_Stack S)

```
{
    D(printf("Pop_Matrix: Top of matrix = %d", S->Top));
    if (STACK_IS_EMPTY(S))
    {
        Non_Fatal_Error("Pop_Matrix",
                        "Cannot pop a matrix - stack is empty");
        /*return NULL_MATRIX;*/
        return NULL;
    }
    else
    {
        D(printf("\n\nPop_Matrix: Top Matrix: \n"));
        D(Put_Matrix(S->Element[S->Top]));

        D(printf("popmatrix successful\n"));
        S->Top--;

        D(printf("ND: Pop_Matrix(): Contents of stack now:\n"));
        D(Put_Matrix_Stack(S));
```

```
        return &(S->Element[S->Top+1]);
    }
}
```

GLboolean Stack_Is_Null(Matrix_Stack The_Stack)

```
{
    return (The_Stack == NULL || The_Stack->Element == NULL);
}
```

void Initialise_Stack(Matrix_Stack *The_Stack, int New_Size)

```
{
   Warning("Initialise_Stack(): Non-essential unction not yet implemented\n");
}
```

void Finalise_Stack(Matrix_Stack The_Stack)

```
/* Purpose: Deallocate the stack */
{
    if (Stack_Is_Null(The_Stack))
    {
        Warning("FinaliseStack: Tried to deallocate a stack that does"
                " not exist");
        return;
    }
    else
    {   /* Deallocate each element */
        int Index;
        for (Index=0; Index < The_Stack->Top; Index++)
            /* Finalise_Stack_Element(The_Stack->Element[Index]);*/
            free(The_Stack->Element[Index]);

        /* Free array of pointers to elements */
        free(The_Stack->Element);
    }
}
```

A.3.4 Support Functionality

support.c

```
/*****************************************************************************
 *
 *
 *           *         n-dimensional graphics library
 *************         ------------------------------
 *************
            *          A. Ellerton
            **         Project for part of BSc(Comp.Sci) Honours
 *************         Edith Cowan University
 ************          (c) A. Ellerton 1995
 *
 *   *****             Component: support.c
 ***********           Description: supporting functions
 ***        **
 *           **
  *          *
 *********************
 *********************
 *
 *

*****************************************************************************/


#include <GL/nd.h>
#include "private.h"
#include <limits.h>           /* For RAND_MAX */
#include <stdarg.h>
```

```
#ifndef RAND_MAX              /* AIX has RAND_MAX in stdlib.h */
#    include <stdlib.h>
#endif
```

void Fatal_Error(const char *Module, const char *Message)

```
{
    fprintf(stderr, "ND FATAL ERROR: Module=>\"%s\": %s\n",
            Module, Message);
    exit(1);
}
```

void Non_Fatal_Error(const char *Module, const char *Message)

```
{
    D(fprintf(stderr, "ND Non-Fatal Error: \n"
                      "    Module => \"%s\"\n"
                      "    Error  => %s\n", Module, Message));
    fprintf(stdout, "ND Non-Fatal Error: \n"
                    "    Module => \"%s\"\n"
                    "    Error  => %s\n", Module, Message);
}
```

float Random_Float(float Minimum, float Maximum)

```
{
    return ((float) rand()/(float) RAND_MAX) * (Maximum-Minimum) + Minimum;
}
```

int Random_Int(int Minimum, int Maximum)

```
{
    return (int) ( ((float) rand()/(float) RAND_MAX) *
        (float)(Maximum-Minimum)) + Minimum;
}
```

void Warning(const char *Format, ...)

```
{
    va_list Arguments;
    va_start(Arguments, Format);        /* Setup Arguments Pointer */

    printf("\nWARNING: ");
    vprintf(Format, Arguments);
    va_end(Arguments);
}
```

A.3.5 Vector Control

vector.c

```
/
*****************************************************************************
 *
 *
 *           *         n-dimensional graphics library
 *************         ------------------------------
 ************
             *         A. Ellerton
             **        Project for part of BSc(Comp.Sci) Honours
 ************        Edith Cowan University
 ************        (c) A. Ellerton 1995
 *
 *   *****           Component: vector.c
 **********          Description: Strictly vector-related functions
 ***       **
 *          **
  *          *
 ********************
 ********************
 *
 *
```

```
*****************************************************************************/

#include <GL/nd.h>
#include "private.h"
#include <stdarg.h>
#include <malloc.h>


/*****************************************************************************
 * TYPES/DEFINITONS ETC

*****************************************************************************/

const Vector NULL_VECTOR = {0, NULL};

/*****************************************************************************
 * VECTORS - AVAILABLE FUNCTIONS

*****************************************************************************/
```

extern Vector Copy_Vector(Vector Vector_To_Copy)

```
{
    Vector Duplicate;
    int Element;

    /*
    if (IS_NULL_VECTOR(Vector_To_Copy))
    */
    if (Vector_To_Copy.Elements==0 ||
        Vector_To_Copy.Element == NULL)
    {
        return NULL_VECTOR;
    }

    /* Allocate */
    New_Vector(Vector_To_Copy.Elements, &Duplicate);

    /* Copy in elements */
    memcpy(Duplicate.Element, Vector_To_Copy.Element,
           Vector_To_Copy.Elements * sizeof(Element_Type));

    DD(printf("Copy_Vector(): Finished copying a vector of %d elements\n",
        Duplicate.Elements));
    return Duplicate;
}
```

extern void Free_Vector(Vector A_Vector)

```
{
   DD(printf("Free_Vector(): Freeing vector of %d elements.\n", A_Vector.Ele-
ments));
/* TEMPORARILY REMOVED
   if (A_Vector.Element != NULL) free (A_Vector.Element);
*/

   ;
}
```

extern void New_Vector(int Number_Of_Elements, Vector *V)

```
{
    DD(printf("New_Vector(): Entered\n"));

    /* 960107 The following sentence is causing seg faults (on SGI's, not AIX).
     *
     * if (V->Element != NULL) free(V->Element);
     *
     * It has been removed, although it _does_ (erroneously) work on AIX.
     */


    if (Number_Of_Elements<=0)
    {
```

```
        *V = NULL_VECTOR;
        return;
    }


    V->Elements = Number_Of_Elements;
    V->Element =
        (Element_Type *) calloc (sizeof(Element_Type), Number_Of_Elements);

    if (V->Element == NULL) /* out of memory */
    {
        Warning("Out of memory when creating a new Vector\n");
        *V = NULL_VECTOR;
        return;
    }

    DD(printf("New_Vector(): Just created new vector with %d elements\n",
      V->Elements));
}
```

Vector Set_Vector(int Number_Of_Elements, ...)

```
/* Create and populate a vector */
{
    Vector V;
    int Element;
    va_list Arguments;

    DD(printf("Set_Vector(): %d elements to allocate ...\n",
          Number_Of_Elements));

    if (Number_Of_Elements <= 0)
    {
        return NULL_VECTOR;
    }

    /* Allocate */
    New_Vector(Number_Of_Elements, &V);
    DD(printf("Set_Vector(): Should've allocated %d elements to vector.\n",
          Number_Of_Elements));

    return V;

    /* Copy in elements */
    va_start(Arguments, Number_Of_Elements);     /* Setup Arguments Pointer */

    for (Element = 1; Element <= V.Elements; Element++)
        VECTOR_ELEMENT(V, Element) = va_arg(Arguments, Element_Type);

    va_end(Arguments);           /* 'clean up' arguments */
    DD(printf("Set_Vector(): Finished setting a vector of %d elements\n",
       V.Elements));
    return V;
}
```

void Put_Vector(Vector V)

```
{
    int Row;

    if (DIMENSION_OF(V) <1 || V.Element == NULL)
    {
        printf(" Null Vector\n");
        return;
    }

    printf("  ( ");
    for (Row=1; Row<=V.Elements; Row++)
    {
        printf("% 1.3f ", VECTOR_ELEMENT(V, Row));
    }
    printf(") \n");
}
```

void Put_Column_Vector(Vector V)

```
{
    int Row;
    for (Row=1; Row<=V.Elements; Row++)
    {
        printf("  %c ", (Row==1)? '(' : ' ');
        printf("% 1.3f    ", VECTOR_ELEMENT(V, Row));
        printf("%c\n", (Row==V.Elements)? ')':' ');
    }
}
```

Appendix B. Source Code: Demonstration Programs

Within this appendix each file, comprising the various demonstration programs built for this project, is listed. There are three sections required:

- *OpenGL*-based programs;
- *Open Inventor*-based programs; and,
- support architecture: for both *OpenGL*- and *Open Inventor*-based programs.

B.1 OpenGL-Based Programs

Each *OpenGL*-based demonstration program is listed below, in alphabetical order, dividing into sections according to their visualisation focus.

B.2 Remote-Sensing Data Visualisation

nd/progs/gl/er1.c

```
/* Purpose: To read in the contents of a .erd file */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdarg.h>
#include <math.h>

#include <GL/nd.h>
#include "support.h"
#include "trackball.h"
#include <glut.h>

const char *Program_Title =
    "-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
=-=-=-\n"
    "      n-dimensional remote-sensing visualisation demonstration (GL Ver-
sion)\n"
    "\n                                  (c) 1996 A. Ellerton \n"
    "-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
=-=-=-\n"
;

const char *Usage_Information =
    "\n\nusage: erl [filename #lines #cells_per_line #bands] ([n | 1 | k}=<di-
mension>}   {X Parameters}\n"
    "\nFor example, NO EXAMPLE AVAILABLE YET!\n"
    "(c) 1996  A. Ellerton\n"
```

```
    *-------------------------------------------------------------------
-----\n\n";

#define STRING_BUFFER_SIZE 80
/* #define D(Anything) */
#define D(Anything) Anything
#define DD(Anything)

#define Finalise_String(A_String) free(A_String)

#define Verify_Allocation(Item,Failure_Message) \
    if (Item == NULL) Memory_Warning(Failure_Message)



int Max_X = 50, Max_Y = 50;

typedef unsigned char Cell_Type;
typedef Cell_Type *Cell_Vector;
typedef Cell_Vector *Cell_Matrix;

typedef struct
{
    char *Value, *Units, *Description;
    float Width;
    Cell_Matrix Cell;
}   Cell_Band;

typedef struct
{
    int Number_Of_Lines;
    int Number_Of_Cells_Per_Line;
    int Number_Of_Bands;
    char *File_Name, *Long_Name;
    Cell_Band *Band;
}   Cell_Dataset;

/* Cell_Dataset Type:

   Cell_Dataset ---< Cell_Band ---| Cell_Matrix ---< Cell_Vector ---< Cell_Type

   That is, a CELL_VECTOR is an array/vector of cells.
            a CELL_MATRIX is an array of vectors, making it a two-dimensional
                array of cells
            a CELL_BAND contains ONE Cell_Matrix. For a multi-band or multi-
                channel dataset, each band has its own array of cells.
            a CELL_DATASET has an array of Cell_Band's, as many as required.
                For example, a 2-band dataset require's an array of
                Cell_Bands[2]

   Layered Bands, diagrammatically,

              /-----------/ <--------- Band 1
             /           /
            /           /-/ <--------- Band 2
           /           / /
          /-----------/ /
           /           /                    :
          /-----------/   :

              /-----------/ <--------- Band n
          :  /        :  /
            /           /
           /           /
          /-----------/

    Each band is a 2D array of Cell_Type:

          Width = Dataset->Number_Of_Cells_Per_Line
          |<------------->|
          |               |

          +---------------+  --
          |               |   |
          |               |   |
          |               |   |
```

```
          |               |   | Height = Dataset->Number_Of_Lines
          |               |   |
          |               |   |
          |               |   |
          +---------------+  --

    The idea of designing the overly hierarchical datastructure is:
    1.each line (Cell_Vector) is often quite long - e.g. 800 cells. It may
      have been easier to write a datastructure which uses one big contiguous
      block of memory for an entire 2D matrix, but at approx 900x800 cells, I
      would rather not risk memory allocation failing.
    2.The ER files are organised so that a vector/line of cells can be read at
      once using (binary) fread(). Although efficiency when file loading is not
      a prime concern, organisation of the datastructure as an array of vectors
      is already useful by (1), and also useful for (2).

    Band 2 of dataset D is accessed by:

        Cell_Dataset D;
        ...
        D.Band[2]

    The 5th line in band 2 of D is:

        D.Band[2].Cell[5]

    The 200th cell of the 5th line in band 2 of D is:

        D.Band[2].Cell[5][200]   <------- I THINK!
*/

/*****************************************************
 * GLOBAL VARIABLES                                  *
 *****************************************************/
Cell_Dataset A_Dataset;
int Width, Height;
GLfloat Rotation_Quaternion[4]={1.0, 0.0, 0.0, 0.0},
        Vector[3] = {0.707, 0.707, 0.707};
int Begin_Mouse_X, Begin_Mouse_Y;



/*****************************************************
 * FUNCTION FORWARD DECLARATIONS                     *
 *
 * (Some missing probably...)
 *****************************************************/
```

void Fatal_Error(const char *Format, ...);
void Memory_Warning(const char *Message);
void Pause(void);

static
void Finalise_ERS_Dataset(Cell_Dataset The_Dataset);

static
Cell_Band Initialise_Band(int Number_Of_Lines,
```
                int Number_Of_Cells_Per_Line,
                const char *Band_Value,
                const char *Band_Units,
                const char *Band_Description,
                float Band_Width);
```

static
Cell_Matrix Initialise_Cell_Matrix(int Number_Of_Lines,
```
                        int Number_Of_Cells_Per_Line);
```

static void
Put_Dataset_Image(Cell_Dataset A_Dataset);

```
void Draw_Raster_Band_Portion_Line3D(Cell_Dataset D, int B, int X1, int Y1,
   int X2, int Y2);

static int
Get_ERS_Line(FILE *F, Cell_Vector V, int N)
{
    int Checksum;

    Checksum = fread(V, 1 /*sizeof(Cell_Type) */, N, F);
    return Checksum;
}


void Graphics_Terminate(void)
{
    Finalise_ERS_Dataset(A_Dataset);
    exit(0);
}


void Graphics_Motion_Handler(int x, int y)
{

  trackball(Rotation_Quaternion,
      (2.0*Begin_Mouse_X-Width)/Width,
      (Height-2.0*Begin_Mouse_Y)/Height,
      (2.0*x -Width)/Width,
      (Height-2.0*y)/Height);
  DD(printf("Q: {%1.3f, %1.3f, %1.3f, %1.3f}\n",
       Rotation_Quaternion[0],
       Rotation_Quaternion[1],
       Rotation_Quaternion[2],
       Rotation_Quaternion[3]));
  glutPostRedisplay();
}

void Graphics_Mouse_Handler(int button, int state, int x, int y)
{
  printf("In Mouse_Handler...\n");
  Begin_Mouse_X = x; Begin_Mouse_Y = y;
}


void Graphics_Draw(void)
{
    GLfloat m[4][4];
    glMatrixMode(GL_MODELVIEW);
    glPushMatrix();

    build_rotmatrix(m, Rotation_Quaternion);
    glMultMatrixf(&m[0][0]);

    Draw_Raster_Band_Portion_Line3D(A_Dataset, 0, 0, 0, Max_X, Max_Y);
    glColor3f(0.8, 0.8, 0.8);
    glutWireCube(1.0);

    glMatrixMode(GL_MODELVIEW);
    glPopMatrix();                  /* Dump non-original matrix */

    glFlush();
}


void Graphics_Keyboard_Handler(char c, int x, int y)
{
    Graphics_Terminate();
}


void Graphics_Reshape_Handler(int w, int h)
{
    glViewport(0, 0, w, h);
```

```
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity ();
    glTranslatef (0.0, 0.0, -3.3);  /*  move object into view   */
    Width = w; Height = h;
}
```

void Load_ERS_Dataset(const char *File_Name,
Cell_Dataset *New_Dataset,
int Number_Of_Lines,
int Number_Of_Cells_Per_Line,
int Number_Of_Bands)

```
/* Purpose: To read in data stored in an ers (binary) file
   Method:
   - Open file
   - Initialise dataset, e.g. # lines, # cells and # bands
   - allocate memory as required for this dataset
   - for each line
        - for each band
            - read a line of data
            - check line was read properly (give an error if not)
   - Inform user of success/failure
 */
{
    FILE *Dataset_File;
    int B, L,
        Checksum;          /* Keeps check of the number of items read */

    /* Setup file for reading */
    Dataset_File = fopen(File_Name, "rb");
    if (Dataset_File == NULL)
    {
       fprintf(stderr, "ER Read Error: Cannot open file \"%s\" for reading.\n",
               File_Name);
        exit(1);
    }

    /* Get Data */
    printf("\nReading ER File... \n"
           "     File:                          \"%s\"\n"
           "     Datatype: 8bit Unsigned Integer\n" /* Should sense this ... */
           "     Number of cells per line (x):  %d\n"
           "     Number of lines (y):           %d\n"
           "     Number of bands:               %d\n",
           File_Name, Number_Of_Cells_Per_Line, Number_Of_Lines,
Number_Of_Bands);

    New_Dataset->Number_Of_Cells_Per_Line = Number_Of_Cells_Per_Line;
    New_Dataset->Number_Of_Lines = Number_Of_Lines;
    New_Dataset->Number_Of_Bands = Number_Of_Bands;
    New_Dataset->File_Name = strdup(File_Name);
    New_Dataset->Long_Name = strdup("Default Dataset Long_Name");

    New_Dataset->Band =
        (Cell_Band *) malloc(sizeof(Cell_Band)*New_Dataset->Number_Of_Bands);
    Verify_Allocation(New_Dataset->Band, "Failed to allocate bands");

    for (B = 0; B < New_Dataset->Number_Of_Bands; B++)
    {
        New_Dataset->Band[B] =
            Initialise_Band(New_Dataset->Number_Of_Lines,
                            New_Dataset->Number_Of_Cells_Per_Line,
                            "Default Band Value",
                            "Default Band Units",
                            "Default Band Description",
                            1.0);
    }

    /* Read the file */
    for (L=0; L<New_Dataset->Number_Of_Lines; L++)
        for (B=0; B<New_Dataset->Number_Of_Bands; B++)
        {
```

```
            DD(printf("Line %d: Band %d: ", L+1, B+1));
          DD(printf("sizeof(Band[%d]=%d  ", B, sizeof(New_Dataset->Band[B])));
            DD(printf("sizeof(Band[%d].Cell=%d   ", B, sizeof(New_Dataset-
>Band[B].Cell) ));
           DD(printf("sizeof(Band[%d].Cell[%d]=%d\n", B,L, sizeof(New_Dataset-
>Band[B].Cell[L]) ));

            Checksum = Get_ERS_Line(Dataset_File,
                                    New_Dataset->Band[B].Cell[L],
                                    New_Dataset->Number_Of_Cells_Per_Line);

            if (Checksum !=  New_Dataset->Number_Of_Cells_Per_Line)
            {
                 Fatal_Error("ERS Reading: Could not read from file or "
                             "EOF encountered unexpectedly");
            }
            DD(printf(" %d Cells \n", Checksum));
        }


    /* Finalise */
    fclose(Dataset_File);

    printf("\nComplete - Read %d lines @ %d cells per line - %d band%c.\n",
           L, New_Dataset->Number_Of_Cells_Per_Line,
           B, (B>1)? 's':'\0');
}
```

void Draw_Raster_Band_Portion_Line3D(Cell_Dataset D, int B, int X1, int Y1, int X2, int Y2)

```
/* Draw band B of dataset D - as lines in 3D */
{
    int i, Row, Col;
    float Gray_Level;
    GLfloat m[4][4];
    static float V[10];

    printf("\nDrawing... %d Bands\n",B);
    ndBegin(GL_POINTS);
    for (Row=Y1; Row<=Y2; Row++)
    {
        for (Col=X1; Col<=X2; Col++)
        {
           Gray_Level = ((float) D.Band[B].Cell[Row][Col])/255.0;
           /*glColor3f(Gray_Level, Gray_Level, Gray_Level);*/
           glColor3f(1.0, 1.0, 0.0);

           /* FORM 1 */
           V[0] = (GLfloat) Col/(GLfloat) (X2-X1);
           V[1] = (GLfloat) Row/(GLfloat) (Y2-Y1);

           for (i=0; i<B; i++)
                V[i+2] = ((GLfloat) D.Band[i].Cell[Row][Col] / 255.0);

           ndVertexNfv(B+1, V);

        /*printf("output %1.1f %1.1f %1.1f\n", (GLfloat) Col/(GLfloat) (X2-X1),
                     ((GLfloat) D.Band[B].Cell[Row][Col] / 255.0),
                    (GLfloat) Row/(GLfloat) (Y2-Y1)
                   ); */
        }
    }
    ndEnd();
}


int n;


int
main(int argc, char **argv)
{
    int Number_Of_Lines, Cells_Per_Line, Number_Of_Bands;
```

```
    Load_ERS_Dataset("models/erland", &A_Dataset, 802, 958, 7);
    n=7;

    Put_Dataset_Image(A_Dataset);


    /* ------------------------------------
       Magic ND Configuration...
       ------------------------------------*/
    Setup_ND_Window(&argc, argv);


    /* ------------------------------------
       Execution
       ------------------------------------*/
    glutMouseFunc(Graphics_Mouse_Handler);
    glutMotionFunc(Graphics_Motion_Handler);

    Manage_ND_Window();

    return 0;
}
```

Fulfil requirements of support module

void Quit(void)

```
{
    /* Finalise_ERS_Dataset(A_Dataset); */
    printf("\n\nER Demonstration Complete.\n");
    exit(0);
}
```

void Setup(void)

```
{
    ;
}
```

void Keyboard(unsigned char The_Key)

```
{
    switch (The_Key)
    {
        case 'x':
            Max_X += 30;
            printf("Horiz Res = %d\n", Max_X);
            glutPostRedisplay();
            break;
        case 'X':
            if (Max_X > 60) Max_X -= 30;
            printf("Horiz Res = %d\n", Max_X);
            glutPostRedisplay();
            break;

        case 'y':
            Max_Y += 30;
            printf("Vert Res = %d\n", Max_Y);
            glutPostRedisplay();
            break;
        case 'Y':
            if (Max_Y > 60) Max_Y -= 30;
            printf("Vert Res = %d\n", Max_Y);
            glutPostRedisplay();
            break;
    }
}
```

void Display(void)

```
{
    GLfloat m[4][4];
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glMatrixMode(GL_MODELVIEW);
    glPushMatrix();
    build_rotmatrix(m, Rotation_Quaternion);
    glMultMatrixf(&m[0][0]);
```

```
    glColor3f(0.8, 0.8, 0.8);

    glColor3f(0.8, 0.8, 0.8);
    glutWireCube(2.0);

    Draw_Raster_Band_Portion_Line3D(A_Dataset, World_Dimension, 0, 0, Max_X,
Max_Y);

    glMatrixMode(GL_MODELVIEW);
    glPopMatrix();                  /* Dump non-original matrix */
}
```

void Reshape(int w, int h)

```
{
    Width = w; Height = h;
    return;
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glMatrixMode(GL_MODELVIEW);
    glTranslatef (0.0, 0.0, -3.3);  /*  move object into view   */
    Width = w; Height = h;

    ndMatrixMode(ND_WORLD_MODELVIEW);
    ndLoadIdentity();
    ndPushMatrix();
}
```

void
Pause(void)

```
{
    printf("\nPress return to continue");
    getchar();
}
```

void Memory_Warning(const char *Message)

```
{
    fprintf(stderr, "WARNING: Out of memory: %s\n", Message);

    /* Other handling as required */
}
```

void Fatal_Error(const char *Format, ...)

```
{
    char Message[STRING_BUFFER_SIZE];
    va_list Arguments;

    va_start(Arguments, Format);
    vsprintf(Message, Format, Arguments);
    va_end(Arguments);

    fprintf(stderr, "\nFATAL ERROR: %s\n", Message);
    Pause();
    exit(1);
}
```

static void Finalise_Cell_Matrix(Cell_Matrix The_Matrix,
int Number_Of_Lines,
int Number_Of_Cells_Per_Line)

```
{
    int Index;

    D(printf("Called Finalise cell matrix (w x h): %d x %d\n",
      Number_Of_Cells_Per_Line, Number_Of_Lines));

    /* Free each line of cells (each "vector") in the array */
    for (Index=0; Index<Number_Of_Lines; Index++)
        free(The_Matrix[Index]);

   /* Free the array of pointers-to-vectors */
```

```
    free(The_Matrix);
}
```

static void Finalise_ERS_Dataset(Cell_Dataset The_Dataset)

```
{
    int Index;

    if (The_Dataset.Band[0].Cell == NULL)
    {
        D(printf("\nCell Dataset already deallocated\n"));
        return;
    }

    D(printf("\nDeallocating the dataset \"%s\"\n", The_Dataset.File_Name));
    Finalise_String(The_Dataset.File_Name);
    Finalise_String(The_Dataset.Long_Name);

    /* Free each band */
    for (Index=0; Index<The_Dataset.Number_Of_Bands; Index++)
    {
        Finalise_String      (The_Dataset.Band[Index].Value);
        Finalise_String      (The_Dataset.Band[Index].Units);
        Finalise_String      (The_Dataset.Band[Index].Description);
        Finalise_Cell_Matrix (The_Dataset.Band[Index].Cell,
                              The_Dataset.Number_Of_Lines,
                              The_Dataset.Number_Of_Cells_Per_Line);
    }

    /* Free array of bands */
    free(The_Dataset.Band);
}
```

static void Put_Dataset_Image(Cell_Dataset A_Dataset)

```
{
    int Index;

    printf("\nDataset Image: \n"
           "    File:                          \"%s\"\n"
           "    Long name:                      %s\n"
           "    Number of cells per line (x):  %d\n" /* Should sense this ... */
           "    Number of lines (y):            %d\n",
           A_Dataset.File_Name,
           A_Dataset.Long_Name,
           A_Dataset.Number_Of_Cells_Per_Line,
           A_Dataset.Number_Of_Lines,
           A_Dataset.Number_Of_Bands);

    for (Index=0;
         Index < A_Dataset.Number_Of_Bands;
         Index++)
    {
        printf("    Band %d:\n"
               "        Value:                    \"%s\"\n"
               "        Units:                    \"%s\"\n"
               "        Description:              \"%s\"\n"
               "        Width:                   %3.3f\n",
               Index+1,
               A_Dataset.Band[Index].Value,
               A_Dataset.Band[Index].Units,
               A_Dataset.Band[Index].Description,
               A_Dataset.Band[Index].Width);
    }
}
```

static Cell_Band
Initialise_Band(int Number_Of_Lines,

```
                  int Number_Of_Cells_Per_Line,
                  const char *Band_Value,
                  const char *Band_Units,
                  const char *Band_Description,
                  float Band_Width)
{
```

```
    Cell_Band New_Band;

    New_Band.Value       = strdup(Band_Value);
    New_Band.Units       = strdup(Band_Units);
    New_Band.Description = strdup(Band_Description);
    New_Band.Width       = Band_Width;
    New_Band.Cell        =
        Initialise_Cell_Matrix(Number_Of_Lines, Number_Of_Cells_Per_Line);
    return New_Band;
}
```

static Cell_Matrix
Initialise_Cell_Matrix(int Number_Of_Lines,
int Number_Of_Cells_Per_Line)

```
{
    Cell_Matrix New_Matrix;
    int Index;

    D(printf("Called init cell matrix (w x h): %d x %d\n",
      Number_Of_Cells_Per_Line, Number_Of_Lines));

    /*  Allocate array of pointers-to-vectors*/
    New_Matrix =
        (Cell_Vector *) malloc(sizeof(Cell_Vector) * Number_Of_Lines);
    Verify_Allocation(New_Matrix, "Failed to allocate matrix of cells");
    DD(printf("sizeof(New_Matrix) = %d\n"
              "sizeof(New_Matrix)/sizeof(Cell_Vector) = %d\n",
              sizeof(New_Matrix),
              sizeof(New_Matrix)/sizeof(Cell_Vector*)));

    /* Allocate each line of cells (each "vector") in the matrix*/
    for (Index=0; Index<Number_Of_Lines; Index++)
    {
        New_Matrix[Index]=
            (Cell_Type *) malloc(sizeof(Cell_Type) * Number_Of_Cells_Per_Line);
        Verify_Allocation(New_Matrix, "Failed to allocate vector of cells");
       DD(printf("sizeof(New_Matrix[%d]) = %d\n", Index, sizeof(New_Matrix[In-
dex])));

    }

    return New_Matrix;
}
```

nd/progs/gl/er2.c

The programs er1 and er2.c are identical except for the function responsible for rendering the datasets. That function is listed below.

void Draw_Raster_Band_Portion_Line3D(Cell_Dataset D, int B, int X1, int Y1, int X2, int Y2)

```
/* Draw band B of dataset D - as lines in 3D */
{
    int i, Row, Col;
    float Gray_Level;
    GLfloat m[4][4];
    static float V[10];\
    printf("\nDrawing... %d Bands\n",B);
    glPointSize(3.0);
    ndBegin(GL_POINTS);
    for (Row=Y1; Row<=Y2; Row++)
    {
        for (Col=X1; Col<=X2; Col++)
        {
           Gray_Level = ((float) D.Band[B].Cell[Row][Col])/255.0;
           /*glColor3f(Gray_Level, Gray_Level, Gray_Level);*/
           glColor3f(1.0, 1.0, 0.0);
```

```
            /* FORM 2 */
            for (i=0; i<B; i++)
                 V[i] = ((GLfloat) D.Band[i].Cell[Row][Col] / 255.0);

            ndVertexNfv(B-1, V);
        }
    }
    ndEnd();
}
```

B.3 Hypercube Visualisation

nd/progs/gl/hcube.c

```
/* Hypercube demo
 * ------------------------------------------------------------------------
 * 951123 (c) A. Ellerton

 * Good planes to try rotating: 4-2 then 3-2 then 1-2.
 *
 */

#include <GL/nd.h>
#include <GL/ndu.h>
#include <glut.h>

#include "polytope.h"
#include "support.h"
#include <stdio.h>
#include <stdlib.h> /* EXIT_SUCCESS etc */

Hyper_Polytope  P;
int             Use_Fog = ND_FALSE;
float           Clear_Colour[4] = {1.0, 1.0, 1.0, 1.0};


const char *Program_Title =
    "-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
=-=-=-\n"
    "                    n-dimensional Hypercube Demonstration (GL Version)\n"
    "\n                                  (c) 1996 A. Ellerton \n"
    "-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
=-=-=-\n"
;

const char *Usage_Information =
    "\n\nusage: hcube {[n | l | k]=<dimension>}  [X Parameters]\n"
    "\nFor example, hcube n=4 k=3 would render a four-dimensional hypercube \n"
    "projected to three-dimensions (no sub-space projection), using OpenGL/In-
ventor\n"
    "for three-dimensional rendering\n\n"
    "(c) 1996  A. Ellerton\n"
    "-------------------------------------------------------------------------
-----\n\n";
```

void Keyboard(unsigned char The_Key)

```
{
    ;
}
```

void Quit(void)

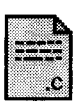
```
{
    Destroy_Polytope(P);
    printf("\n\nHypercube Demonstration Complete.\n");
    exit(0);
}
```

```
void Setup(void)
{
    if (Use_Fog)
    {
        glEnable(GL_FOG);
        glFogi (GL_FOG_MODE, GL_LINEAR);
        glHint (GL_FOG_HINT, GL_NICEST);  /*  per pixel   */
        glFogf (GL_FOG_START, 2.0);
        glFogf (GL_FOG_END, 5.0);
        glFogfv (GL_FOG_COLOR, Clear_Colour);

        glDepthFunc(GL_LESS);
        glEnable(GL_DEPTH_TEST);
        glShadeModel(GL_FLAT);
    }
    glClearColor(Clear_Colour[0], Clear_Colour[1],
                 Clear_Colour[2], Clear_Colour[4]);
}
```

```
void Display(void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glColor3f(1.0, 0.0, 0.0);
    Render_Polytope(P);
}
```

```
void Reshape(int w, int h)
[
    ;
}
```

```
int main(int argc, char **argv)
(

    /* ------------------------------------
       Magic ND Configuration...
       ------------------------------------*/
    Setup_ND_Window(&argc, argv);


    /* ------------------------------------
       Object Configuration
       ------------------------------------*/

    P = Generate_Hypercube(World_Dimension);
    Display_Polytope_Information(P);

    /* ------------------------------------
       Execution
       ------------------------------------*/
    Manage_ND_Window();
    return EXIT_SUCCESS;
}
```



nd/progs/gl/polytope.c

```
#include <GL/nd.h>

#include "polytope.h"
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define D(Anything) Anything
#define DD(Anything)

#define DRAW_FACES_IDENTICALLY 1
```

```
#define DRAW_HIGH_DIM_FACE_THICK  2

/* FLAGS */

int Line_Thickness_Flag = DRAW_HIGH_DIM_FACE_THICK;
/*int Line_Thickness_Flag = DRAW_FACES_IDENTICALLY;*/
int Line_Stipple_Flag   = ND_TRUE;


int LINE_STIPPLE = 0xCCCC;
float Thin_Line_Thickness = 2.0;
float Thick_Line_Thickness = 3.0;
```

Vertex New_Vertex(int Dimension)

```
{
    Vertex The_Vertex;

    The_Vertex = (Vertex) calloc (sizeof(float),Dimension);
    if (The_Vertex == NULL)
    {
        fprintf(stderr, "Out of memory");
        getchar();
        exit(1);
    }
    return The_Vertex;
}
```

Vertex_List New_Vertex_List(int Dimension,
int Vertex_Count)

```
{
    int Vertex_Index;
    Vertex_List The_Vertex_List;

    The_Vertex_List = (Vertex_List) calloc (sizeof(Vertex), Vertex_Count);
    for (Vertex_Index=0;
         Vertex_Index<Vertex_Count;
         Vertex_Index++)
         The_Vertex_List[Vertex_Index] = New_Vertex(Dimension);
    return The_Vertex_List;
}
```

Boolean_Matrix
New_Boolean_Matrix(int Rows, int Cols)

```
{
    int Row;
    Boolean_Matrix The_Boolean_Matrix;

    The_Boolean_Matrix = (Boolean **) malloc (sizeof(Boolean *) * Rows);
    for (Row=0; Row<Rows; Row++)
    {
        The_Boolean_Matrix[Row] = (Boolean *) calloc (sizeof(Boolean), Cols);
    }
    D(printf("Allocated %d x %d boolean matrix\n", Rows, Cols));
    return The_Boolean_Matrix;
}
```

void Finalise_Vertex_List(Vertex_List The_Vertex_List)

```
/* Free memory allocated for list of vertices.
   Method: Loop through each vertex
           Deallocate each vertex
           Deallocate VertexList memory
 */
{
    ;
}
```

void Destroy_Polytope(Hyper_Polytope P)

```
{
    /* Free dynamically allocated memory for this object */
    ;
```

```
}
```

float Distance(int N, Vertex A, Vertex B)

```
{
   int i;
   double Total = 0.0;

   for (i=0; i<N; i++) {
       Total += (A[i]-B[i]) * (A[i]-B[i]);
   }
   return (float) sqrt(Total);
}
```

int Power(int x, int y)

```
/* return x to the power y */
{
   int Result=1;

   if (y < 0)
       return 0; /* Not handling neg powers */

   while (y > 0)
   {
      Result *= x;
      y -= 1;
   }
   return Result;
}
```

Hyper_Polytope Generate_Hypercube(int n)

```
/* Generates a n-dimensional hypercube */
{
    int Vertices, Edges;
    int Vertex_Index, Element_Index, Alternate;
    int From_Vertex, To_Vertex;
    float Current_Value;
    Vertex_List A_Vertex_List;
    Boolean_Matrix An_Edge_List;
    Hyper_Polytope The_Hypercube;

    if (n < 2)
    {
        fprintf(stderr, "Cannot generate a hypercube of less than 2 dimen-
sions\n");
        return;
    }

    /* SETUP */
    Vertices = Power(2, n);
    Edges = 0;
    A_Vertex_List = New_Vertex_List(n, Vertices);
    An_Edge_List  = New_Boolean_Matrix(Vertices,Vertices);
    Alternate = 4;
    Alternate = Vertices / 2;
    Current_Value = 1.0;

    /* FEEDBACK */
    printf("\nGenerating %d-dimensional hypercube\n"
           "    %d Vertices\n"
           "    %d Edges\n", n, Vertices, Edges);

    /* GENERATE VERTICES */
    for (Element_Index=0;
         Element_Index<n;
         Element_Index++)
    {
        for (Vertex_Index=1;
             Vertex_Index<=Vertices;
             Vertex_Index++)
        {
            A_Vertex_List[Vertex_Index-1][Element_Index] = Current_Value;
            if (Vertex_Index % Alternate == 0 || Alternate == 1 )
```

```
                Current_Value = -Current_Value;
            }
            Alternate = Alternate / 2;
        }

        /* CREATE EDGE ARRAY */
        /* Current_Edge = 1; Check_Vertex = 1; Current_Vertex = 1; */
        for (From_Vertex=0; From_Vertex < Vertices; From_Vertex++) {
            DD(printf("Edge from %d ", From_Vertex));
            /* 951123: changed TO_Vertex = 0 */
            for (To_Vertex=From_Vertex; To_Vertex < Vertices; To_Vertex++) {
                if (To_Vertex != From_Vertex &&
                    Distance(n,
                             A_Vertex_List[From_Vertex],
                             A_Vertex_List[To_Vertex]) == 2.0)
                {
                    DD(printf("->%d ", To_Vertex));
                    An_Edge_List[From_Vertex][To_Vertex] = 1;
                } else {
                    An_Edge_List[From_Vertex][To_Vertex] = 0;
                }
            }
            DD(printf("\n"));
        }


        The_Hypercube.Vertices = A_Vertex_List;
        The_Hypercube.Edge     = An_Edge_List;
        The_Hypercube.Dimension= n;
        The_Hypercube.Number_Of_Vertices = Vertices;
        return The_Hypercube;
    }
```

void Display_Polytope_Information(Hyper_Polytope P)

```
{
    int From_Vertex, To_Vertex, Vertex_Index, Element_Index;

    /* FEEDBACK VERTEX LIST */
    printf("DISPLAY POLYTOPE INFO CALLED\n");

    for (Vertex_Index=0; Vertex_Index<P.Number_Of_Vertices; Vertex_Index++)
    {
         printf("Vertex %d:\t[", Vertex_Index+1);
         for (Element_Index=0; Element_Index<P.Dimension; Element_Index++)
             printf("% 1.3f   ",P.Vertices[Vertex_Index][Element_Index]);
         printf("]\n");
    }

    /* FEEDBACK EDGE LIST */
    for (From_Vertex=0; From_Vertex < P.Number_Of_Vertices; From_Vertex++)
    {
        for (To_Vertex=0; To_Vertex < P.Number_Of_Vertices; To_Vertex++)
            printf("%d  ", P.Edge[From_Vertex][To_Vertex]);
        printf("\n");
    }
}
```

void Render_Polytope(Hyper_Polytope P)

```
{
    int From_Vertex, To_Vertex, Vertex_Index, Element_Index;

    for (From_Vertex=0; From_Vertex < P.Number_Of_Vertices; From_Vertex++)
        for (To_Vertex=From_Vertex+1;
             To_Vertex<P.Number_Of_Vertices; To_Vertex++)

            if (P.Edge[From_Vertex][To_Vertex] == 1)
            {
                if (P.Vertices[To_Vertex][P.Dimension-1] > 0 &&
                    Line_Thickness_Flag == DRAW_HIGH_DIM_FACE_THICK)
                {
                    glColor3f(0.0,0.0,0.0);
                    glLineWidth(Thick_Line_Thickness);
                }
```

```
                else
                {
                    glColor3f(0.0,0.0,0.0);
                    /*glColor3f(0.2,0.2,0.2);*/
                    glLineWidth(Thin_Line_Thickness);
                }
/*
                if (Line_Stipple_Flag == ND_TRUE)
                {
                    glEnable(GL_LINE_STIPPLE);
                    glLineStipple(3, LINE_STIPPLE);
                }
*/

                /* IF x4 component < 0 turn dashed lines on */
                if (P.Vertices[To_Vertex][3] <0 ||
                    P.Vertices[From_Vertex][3] < 0)
                    if (Line_Stipple_Flag == ND_TRUE)
                    {
                        glEnable(GL_LINE_STIPPLE);
                        glLineStipple(1, LINE_STIPPLE);
                    }



                ndBegin(GL_LINES);
                ndVertexNfv(P.Dimension, P.Vertices[From_Vertex]);
                ndVertexNfv(P.Dimension, P.Vertices[To_Vertex]);
                ndEnd();
                glDisable(GL_LINE_STIPPLE);

            }
}
```

B.4 Visualisation of Klein Bottle and Car

The klein bottle is constructed mathematically, while the description of the volkswagon is read from a file; this file is not listed here, but is available from the author.

nd/progs/gl/klein.c

Although the *OpenGL*-based Klein bottle program works, the images from the Inventor-based program were more useful, and hence used in chapter five. The code for the *Inventor*-based example is based on the code for the *OpenGL* one, and hence only that example is listed, see Section B.7.

nd/progs/gl/vw.c

```
/* Volkswagon demo
 * ------------------------------------------------------------------------
 * 951123 (c) A. Ellerton
 *
 */

#include <GL/nd.h>
#include <GL/ndu.h>
#include <glut.h>

#include "graphicObject.h"
#include "support.h"
```

```
#include "polytope.h"
#include <stdio.h>
#include <stdlib.h> /* EXIT_SUCCESS etc */

MGLUgraphicObject Model;
Hyper_Polytope Cube;
int             Use_Fog = ND_FALSE;
float           Clear_Colour[4] = {1.0, 1.0, 1.0, 1.0};


static double Scale_Half_Vector[] = {0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5};
static double Scale_Cube_Vector[] = {1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0};
static double Scale_View_Vector[] = {1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0};
static double Scale_Car_Vector[]  =  {2.0, 2.0, 2.0, 1.0, 1.0, 1.0, 1.0};

const char *Program_Title =
    "-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
=-=-=-\n"
    "                  n-dimensional Volkswagon Demonstration (GL Version)\n"
    "\n                               (c) 1996 A. Ellerton \n"
    "-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
=-=-=-\n"
;

const char *Usage_Information =
    "\n\nusage: vw ([n | 1 | k]=<dimension>}  [X Parameters]\n"
   "\nFor example, vw n=4 k=3 would render a three-dimensional volkswagon, but
\n"
    "within four-dimensional space. \n\n"
    "(c) 1996  A. Ellerton\n"
    "----------------------------------------------------------------------
---\n\n";
```

void Keyboard(unsigned char The_Key)

```
{
    ;
}
```

void Quit(void)

```
{
    mgluDestroyGraphicObject(Model);
    printf("\n\nVolksy Demonstration Complete.\n");
    exit(0);
}
```

void Setup(void)

```
{
    mgluCreateGraphicObjectFromFile(&Model,
        "models/vw.geo", "models/vw.pno", NULL);
    Cube = Generate_Hypercube(World_Dimension);
    ndMatrixMode(ND_WORLD_MODELVIEW);
    ndScale(World_Dimension, &Scale_View_Vector[0]);

    if (Use_Fog)
    {
        glEnable(GL_FOG);
        glFogi (GL_FOG_MODE, GL_LINEAR);
        glHint (GL_FOG_HINT, GL_NICEST);  /*  per pixel   */
        glFogf (GL_FOG_START, 2.0);
        glFogf (GL_FOG_END, 5.0);
        glFogfv (GL_FOG_COLOR, Clear_Colour);

        glDepthFunc(GL_LESS);
        glEnable(GL_DEPTH_TEST);
        glShadeModel(GL_FLAT);
    }
    glClearColor(Clear_Colour[0], Clear_Colour[1],
                 Clear_Colour[2], Clear_Colour[4]);


}
```

```
void Display(void)
{
    int Drawing_Flags;
    Drawing_Flags = MGLU_RENDER_EDGES;
    /*  Drawing_Flags = MGLU_RENDER_FACES;*/

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    ndMatrixMode(ND_WORLD_MODELVIEW);
    ndPushMatrix();
        ndScale(World_Dimension, &Scale_Cube_Vector[0]);
        Render_Polytope(Cube);
    ndPopMatrix();

    glColor3f(0.0, 0.0, 0.0);
    ndMatrixMode(ND_WORLD_MODELVIEW);
    ndPushMatrix();
        ndScale(World_Dimension, &Scale_Car_Vector[0]);
        mgluRenderGraphicObject(Model, Drawing_Flags);
    ndPopMatrix();
}

void Reshape(int w, int h)
{
    ;
}


int main(int argc, char **argv)
{
    /* ------------------------------------
       Magic ND Configuration...
       ------------------------------------*/
    Setup_ND_Window(&argc, argv);




    /* ------------------------------------
       Execution
       ------------------------------------*/

    Manage_ND_Window();
    return EXIT_SUCCESS;
}
```

2.5 Common Support Module

nd/progs/gl/support.h

```
/*****************************************************************************
 *
 *
 *           *         n-dimensional graphics library
 ************         ------------------------------
 ************
            *         A. Ellerton
            **        Project for part of BSc(Comp.Sci) Honours
 ************         Edith Cowan University
 ***********          (c) A. Ellerton 1995
 *
 *   *****            Component: support.h
 ***********          Description: general support functions
 ***        **
 *           **
  *          *
 *********************
 *********************
 *
 * ------------------
```

```
 * Uses your functions Quit, Display and Reshape (for those functions)
 *
*****************************************************************************/


#ifndef SUPPORT_H
#define SUPPORT_H

#include <glut.h>

#define TO_RADIANS(angle) ((angle)*M_PI/180.0)

extern int World_Dimension, Subspace_Dimension, Device_Dimension;
extern const char *Program_Title, *Usage_Information;
```

Functions Module Must Supply

```
extern void Quit();                    /* YOUR Quit function */
extern void Setup();                   /* YOUR Setup (eg ndDimension) function */
extern void Display();                 /* YOUR Display function */
extern void Reshape(int w, int h);     /* YOUR reshape function */
extern void Keyboard(unsigned char The_Key);    /* YOUR keyboard function */
```

Available Functions

```
extern void Setup_ND_Window();
extern void Manage_ND_Window();
```

Default Event Handlers

```
extern void Handle_Display_Event(void);
extern void Handle_Mouse_Event(int Button, int State, int x, int y);
extern void Handle_Motion_Event(int x, int y);
extern void Handle_Keyboard_Event(unsigned char The_Key, int x, int y);
extern void Handle_Idle( void );

#endif
```

nd/progs/gl/support.c

```
/* Support Functionality
 * ----------------------------------------------------------------------
 * 951123 (c) A. Ellerton
 *
 */


#include "support.h"

#include <GL/nd.h>
#include <GL/ndu.h>

#include <GL/gl.h>
#include <GL/glu.h>
#include "glut.h"

#include <stdio.h>
#include <string.h>                    /* strncmp etc */
#include <stdlib.h>                    /* EXIT_SUCCESS etc */

/* GLOBAL VARIABLES */
int World_Dimension    = 3,
    Subspace_Dimension = 0,
    Device_Dimension   = 2;
static int Subspace_Projection = ND_FALSE;

const int MAXIMUM_DIMENSION = 12;


/* STATIC GLOBAL VARIABLES */

static NDXContext Context;
static GLfloat fogColor[4] = {0.0, 0.0, 0.0, 1.0};

static float Rotation_Angle = 1.0;
static int A=1, B=2; /* Rotation plane */
static float r, f;
```

```
const static float Position_Step = 0.1;

static double Zoom_Out[] =
    {0.99, 0.99, 0.99, 0.99, 0.99, 0.99, 0.99, 0.99, 0.99, 0.99, 0.99, 0.99};
static double Zoom_In[] =
    {1.01, 1.01, 1.01, 1.01, 1.01, 1.01, 1.01, 1.01, 1.01, 1.01, 1.01, 1.01};
static double Move_Further_On_4th_Axis[] = {0.0, 0.0, 0.0, 0.1};
static double Translation_Vector[] =
    {0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0};
static Width, Height;
```

void Handle_Idle(void)

```
{
  glutPostRedisplay();
}
```

void Handle_Keyboard_Event(unsigned char The_Key, int x, int y)

```
{
    static int Animate = ND_FALSE;
    static int Enter_Value = ND_FALSE;
    static int *Dimension = &World_Dimension;
    static NDenum Matrix_Mode = ND_WORLD_MODELVIEW;

    if (The_Key == 'P')
    {
        Enter_Value = ND_TRUE;
        printf("Manual entry mode, which parameter? \n");
    }

    switch (The_Key)
    {

        case 'q':
        case 'Q':
            Quit();
            break;

        case '\\':

            /* TOGGLE MATRIX ACCESS */
            Matrix_Mode = (Matrix_Mode == ND_WORLD_MODELVIEW)?
                ND_SUBSPACE_MODELVIEW : ND_WORLD_MODELVIEW;

            printf("Matrix Mode set to %s\n",
                (Matrix_Mode == ND_WORLD_MODELVIEW)?
                    "ND_WORLD_MODELVIEW" : "ND_SUBSPACE_MODELVIEW"
            );
            break;

        case '|':
            /* TOGGLE SUBSPACE PROJECTION */
            if (Subspace_Projection)
            {
                ndDisable(ND_SUBSPACE_PROJECTION);
                Subspace_Projection = ND_FALSE;
            }
            else
            {
                ndEnable(ND_SUBSPACE_PROJECTION);
                Subspace_Projection = ND_TRUE;
            }
            printf("Subspace projection set to %s\n",
                (Subspace_Projection)? "ON" : "OFF");
            break;


        case '1':
        case '2':
        case '3':
        case '4':
        case '5':
        case '6':
        {   int x = The_Key - '0'; /* x is potenial new A */
```

```
    if (x != B && x != A && x <= World_Dimension)
    {
        A = x;
        printf("Plane is now %d-%d\n", A, B);
    }
    break;
}

case 'a':
case 'b':
case 'c':
case 'd':
case 'e':
case 'g':
{   int x = The_Key - 'a' + 1; /* x is potenial new B */
    if (x != B && x != A && x <= World_Dimension)
    {
        B = x;
        printf("Plane is now %d-%d\n", A, B);
    }
    break;
}

case ' ':
case '-':
    ndMatrixMode(Matrix_Mode);
    ndRotate (*Dimension, A, B,
         (The_Key == ' ')? Rotation_Angle : -Rotation_Angle);
    glutPostRedisplay();
    break;

case 'i':
    ndMatrixMode(Matrix_Mode);
    ndPopMatrix();
    /*ndLoadIdentity();*/
    ndPushMatrix();
    break;

/* Decrease value of f, the hyperplane position on nth axis */
case 'f':
    if (Enter_Value == ND_TRUE)
    {
        printf("Enter value for f: ");
        if (scanf("%f", &f)) printf("Set f to % 1.3f\n", f);
        Enter_Value = ND_FALSE;
    }
    else
    {
        f -= Position_Step;
    }
    ndSet(ND_HYPERPLANE_POSITION, f);
    glutPostRedisplay();
    printf("Hyperplane Position set to %1.3f.\n", f);
    break;

/* Increase value of f, the hyperplane position on nth axis */
case 'F':
    f += Position_Step;
    ndSet(ND_HYPERPLANE_POSITION, f);
    glutPostRedisplay();
    printf("Hyperplane Position set to %1.3f.\n", f);
    break;

/* Decrease value of r, the observers position on nth axis */
case 'r':
    if (Enter_Value == ND_TRUE)
    {
        printf("Enter value for r: ");
        if (scanf("%f", &r)) printf("Set r to % 1.3f\n", r);
        Enter_Value = ND_FALSE;
    }
    else
    {
        r -= Position_Step;
    }
    ndSet(ND_OBSERVER_POSITION, r);
```

```
            glutPostRedisplay();
            printf("Observer Position set to %1.3f.\n", r);
            break;

        /* Increase value of r, the observers position on nth axis */
        case 'R':
            r += Position_Step;
            ndSet(ND_OBSERVER_POSITION, r);
            glutPostRedisplay();
            printf("Observer Position set to %1.3f.\n", r);
            break;

        case 't':
        case 'T':
            /* Translate in direction of axis denoted by A */
            Translation_Vector[A-1] +=
                (Position_Step * ((The_Key == 'T')? 1.0 : -1.0 ));
            glutPostRedisplay();
            printf("Translation along axis %d is now % 1.3f\n",
                A, Translation_Vector[A-1]);
            break;


        case 'p':
        {   /* Allow user to pick rotation plane */
            int a, b;
            printf("Enter new rotation plane (eg 1 2): ");
            if (scanf("%d %d", &a, &b))
            {
                printf("Set rotation plane to %d-%d\n", a,b);
                A = a; B = b;
            }
            else
            {
                printf("Didn't set rotation plane\n");
            }
            break;
        }


        case 's':
        case 'S':
        {
            ndMatrixMode(Matrix_Mode);
            ndScale(*Dimension, (The_Key == 's')? Zoom_Out : Zoom_In);
            glutPostRedisplay();
            break;
        }


        default:
            Keyboard(The_Key);
            /*printf("Char = %c = %d\n", The_Key, The_Key);*/
            break;
    }
}
```

void Handle_Display_Event(void)

```
{
    /* Effect_Modelling_Transformations */
    ndMatrixMode(ND_WORLD_MODELVIEW);
    ndPushMatrix();
    ndTranslate(World_Dimension, Translation_Vector);

    Display(); /* USERS function */

    ndPopMatrix();


    glutSwapBuffers(); /* WAS glFlush(); */
}
```

void Handle_Mouse_Event(int Button, int State, int x, int y)

```
{
    ;
}
```

void Handle_Motion_Event(int x, int y)

```
{
    ;
}
```

void Handle_Reshape_Event(int w, int h)

```
{
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective (45.0, (GLfloat) w/(GLfloat) h, 0.2, 8.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity ();
    glTranslatef (0.0, 0.0, -3.5);  /*  move object into view   */
    glPushMatrix();                 /*  setup ready for translations... */

    Reshape(w, h);

    Width = w; Height = h;
}
```

void Setup_ND_Window(int *argc, char **argv)

```
{
    int i;


    /* ------------------------------------
       Parse parameters
       ------------------------------------*/
    if (*argc==1 || strcmp(argv[1], "-help") == 0)
    {
        printf("%s%s", Program_Title, Usage_Information);
        exit(0);
    }
    else
    {
        for (i=1; i< *argc; i++)
        {
            if (strncmp(argv[i], "n=", 2) == 0)
                World_Dimension = atoi(&argv[i][2]);

            else if (strncmp(argv[i], "l=", 2) == 0)
                Subspace_Dimension = atoi(&argv[i][2]);

            else if (strncmp(argv[i], "k=", 2) == 0)
                Device_Dimension = atoi(&argv[i][2]);

        }
    }

    /* ------------------------------------
       Glut/Windowing Configuration
       ------------------------------------*/
    glutInit(argc, argv);
    glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
    glutInitWindowPosition(500,500);
    glutInitWindowSize(620,480);
    glutCreateWindow(argv[0]);

    /* ------------------------------------
       ND Configuration
       ------------------------------------*/
    Context = ndXCreateContext();
    ndXMakeCurrent(Context);
    ndGetfv(ND_HYPERPLANE_POSITION, &f);
    ndGetfv(ND_OBSERVER_POSITION,   &r);

    glutDisplayFunc( Handle_Display_Event);
    glutReshapeFunc (Handle_Reshape_Event);
```

```
    glutMouseFunc    (Handle_Mouse_Event);
    glutMotionFunc   (Handle_Motion_Event);
    glutKeyboardFunc(Handle_Keyboard_Event);
    /*glutIdleFunc     (Handle_Idle);*/


    /* ------------------------------------
       GL Configuration
       ------------------------------------*/

    glEnable(GL_FOG);
    glFogi (GL_FOG_MODE, GL_LINEAR);
    glHint (GL_FOG_HINT, GL_NICEST);  /*  per pixel   */
    glFogf (GL_FOG_START, 4.0);
    glFogf (GL_FOG_END, 5.0);
    glFogfv (GL_FOG_COLOR, fogColor);

    glDepthFunc(GL_LESS);
    glEnable(GL_DEPTH_TEST);
    glShadeModel(GL_FLAT);

   /* glClearColor(0.95, 0.95, 0.95, 0.95); /* Slightly off-white, prints ok */
    glClearColor(1.0, 1.0, 1.0, 1.0); /* White */

    Setup();
}
```

void Manage_ND_Window(void)

```
{
    printf("%s\n\tWorld Dimension => %d\n"
           "\tSubspace Dimension => %d\n"
           "\tDevice Dimension => %d\n",
           Program_Title,
           World_Dimension, Subspace_Dimension, Device_Dimension);

    /* WORLD SPACE SWITCHES */
    ndDimension(ND_WORLD_DIMENSION,  World_Dimension);

    /* SUBSPACE SWITCHES */
    if (Subspace_Projection == ND_FALSE ||
        Subspace_Dimension == World_Dimension || Subspace_Dimension == 0)
    {
        printf("\tSubspace Projection: Disabled\n");
        ndDisable(ND_SUBSPACE_PROJECTION);
        if (World_Dimension == Device_Dimension) {
            printf("\tWarning: Because the device dimension is the same as the"
               "\n\tworld dimension, some parameters (such as r and f) may not\t"
                   "\tfunction very well.\n");
        }
    }
    else
    {
        printf("\tSubspace Projection: Enabled at %dD\n", Subspace_Dimension);
        ndDimension(ND_SUBSPACE_DIMENSION, Subspace_Dimension);
        ndEnable(ND_SUBSPACE_PROJECTION);
    }

    /* DEVICE SWITCHES */
    ndDimension(ND_DEVICE_DIMENSION, Device_Dimension);
    ndHint(ND_USE_GL_HINT, ND_IGNORE_GL);

    glutMainLoop();
}
```

B.6 Building Instructions

nd/progs/gl/Imakefile

```
/* Based on the Imake file by Mark J. Kilgard (for GLUT)
 *
 * A.Ellerton 1995
```

```
 */

/* NO subdirs */

#include <Library.tmpl>

#include "../../ND.cf"
/*#include "$(TOPDIR)/ND.cf"*/


OPENGL_GLUT_Target(klein3d, klein3d.o)
OPENGL_GLUT_Target(mobius, mobius.o)
OPENGL_GLUT_Target(cube3d, cube3d.o)
OPENGL_GLUT_Target(cube4d_using_GL, cube4d_using_GL.o)
OPENGL_GLUT_Target(er3d, er3d.o support.o trackball.o )

ND_GLUT_Target(klein4d, klein4d.o)
ND_GLUT_Target(cube, cube.o)
ND_GLUT_Target(hcube, hcube.o polytope.o support.o)
ND_GLUT_Target(klein, klein.o support.o)
ND_GLUT_Target(vw, vw.o support.o graphicObject.o polytope.o)
ND_GLUT_Target(er1, er1.o support.o trackball.o )
ND_GLUT_Target(er2, er2.o support.o trackball.o )
ND_GLUT_Target(mobius4d, mobius4d.o support.o trackball.o )
ND_GLUT_Target(grid, grid.o support.o trackball.o )

DependTarget()
```

B.7 *Open Inventor*-Based Programs

Although several *Inventor*-based demonstration programs were under development, only `klein.c++` is complete enough for listing.

B.8 Visualisation of Klein Bottle

nd/progs/inventor/klein.c++

```
/*****************************************************************************
 ND LIBRARY: KLEIN BOTTLE DEMONSTRATION
 ----------------------------------------------------------------------------
 TO COMPILE: (On SGI)
 /usr/bin/CC -o klein -xansi  -nostdinc                                      \
     -I../../include  -I/usr/include/CC -I/usr/include -O                    \
     -MDupdate Makedepend klein.c++ -quickstart_info  -nostdlib              \
     -L/usr/lib  -lGLU -lGL -lX11 -lm  -lInventorXt


 ----------------------------------------------------------------------------
 DIARY
 960120. Wrote initial version - alloc/dealloc/initial assignment of array is
         correct. No GL integration yet. I'm still working on the mathematica
         version to ensure, a little bit, that Hartley's coordinates are the
         right ones.

 960124. Seems to be working; ported to use inventor and C++.
         Inventor stuff based on quads demo, an Inventor Mentor example.

         Note that the quad must be specified in row major order.

*****************************************************************************/



/* ------------------------------------------------------------------------- *
 * Resource Inclusion
```

```
 * ------------------------------------------------------------------------ */
#include <stdio.h>
#include <math.h>
#include <GL/nd.h>

#include <Inventor/Xt/SoXt.h>
#include <Inventor/Xt/viewers/SoXtExaminerViewer.h>
#include <Inventor/nodes/SoCoordinate3.h>
#include <Inventor/nodes/SoEventCallback.h>
#include <Inventor/events/SoKeyboardEvent.h>
#include <Inventor/nodes/SoMaterial.h>
#include <Inventor/nodes/SoQuadMesh.h>
#include <Inventor/nodes/SoSeparator.h>
#include <Inventor/nodes/SoSelection.h>



////////////////////////////////////////////////////////////////////////////////
//
// Macros
////////////////////////////////////////////////////////////////////////////////
//


#ifdef TWO_PI
#    undef TWO_PI
#endif
#define TWO_PI (2.0 * M_PI)


#define CHECK_ALLOCATION(Item, Message) \
    if(Item==NULL) fprintf(stderr, "%s: OUT OF MEMORY: %s, Line %d\n", \
        __FILE__, Message, __LINE__);


#define ADJACENT(i) (i==Subdivisions-1)? 0 : ((i)+1)

#ifndef EXIT_SUCCESS
#   define EXIT_SUCCESS 0
#endif


////////////////////////////////////////////////////////////////////////////////
//
// Global Variables
////////////////////////////////////////////////////////////////////////////////
//

NDXContext      Context;
SoQuadMesh      *myQuadMesh;
SoCoordinate3   *Bottle_Coords;

int             Strip_Display = 0; // 1 == TRUE, 0 == FALSE
int             n;
typedef float * Vector;
Vector **       Bottle;
const           Subdivisions = 26;
const int       World_Dimension = 4;
const int       Device_Dimension = 3;

int             Use_Material = ND_TRUE; // if true, uses gold-ish colouring
SoMaterial    * myMaterial;

////////////////////////////////////////////////////////////////////////////////
//
// Local Function prototypes
////////////////////////////////////////////////////////////////////////////////
//

void myKeyPressCB(void *, SoEventCallback *);

void Create_Klein_Bottle(float Radius);
void Destroy_Klein_Bottle(void);
void Draw_Klein_Bottle(void);
```

```
// Routine to create a scene graph representing an arch.
SoSeparator *
makeArch()
{
   SoSeparator *result = new SoSeparator;
   result->ref();

   // Define the material
   myMaterial = new SoMaterial;
   if (Use_Material)
   {
       myMaterial->diffuseColor.setValue(.78, .57, .11);
       myMaterial->ambientColor.setValue(0.3, 0.1, 0.1);
       myMaterial->specularColor.setValue(0.4, 0.3, 0.1);
       //myMaterial->transparency.setValue(0.5);
   }
   else
   {
       myMaterial->diffuseColor.setValue(0.0, 0.0, 0.0); // for wireframe only
   }
   result->addChild(myMaterial);

   // Define coordinates for vertices
   Bottle_Coords = new SoCoordinate3;



   myQuadMesh = new SoQuadMesh;
   Draw_Klein_Bottle();

   result->addChild(Bottle_Coords);
   result->addChild(myQuadMesh);

   result->unrefNoDelete();
   return result;

}



void
main(int argc, char **argv)
{
    ////////////////////////////////////////////////////////////////////////
    // Initialise ND Context
    //

    Context = ndXCreateContext();
    ndXMakeCurrent(Context);
    ndSet(ND_HYPERPLANE_POSITION, 3.5);
    ndSet(ND_OBSERVER_POSITION,   3.5);
    ndDimension(ND_WORLD_DIMENSION, World_Dimension);
    ndDimension(ND_DEVICE_DIMENSION, Device_Dimension);
    ndDisable(ND_SUBSPACE_PROJECTION);



    ////////////////////////////////////////////////////////////////////////
    // Local Configuration
    //
    n = 4;
    Create_Klein_Bottle(2.0);



    ////////////////////////////////////////////////////////////////////////
    // Initialize Inventor and Xt
    //
    Widget myWindow = SoXt::init(argv[0]);
    if (myWindow == NULL) exit(1);

    SoSeparator *root = makeArch();
```

```
    root->ref();


    ////////////////////////////////////////////////////////////////////////
    // Install Callbacks
    // An event callback node so we can receive key press events

    SoEventCallback *myEventCB = new SoEventCallback;
    myEventCB->addEventCallback(
            SoKeyboardEvent::getClassTypeId(),
            myKeyPressCB, root);
    root->addChild(myEventCB);


    ////////////////////////////////////////////////////////////////////////
    SoXtExaminerViewer *myViewer =
            new SoXtExaminerViewer(myWindow);
    myViewer->setSceneGraph(root);
    myViewer->setTitle("ND: Klein Bottle Demonstration");
    //myViewer->setBackgroundColor(SbColor(0.95,0.95,0.95));
    myViewer->setBackgroundColor(SbColor(1.00,1.00,1.00));
    myViewer->show();
    myViewer->viewAll();

    SoXt::show(myWindow);
    SoXt::mainLoop();
}

// From 10.1.addEventCB.c++
//
// If the event is down arrow, then remove one row from the quadmesh.
// if the event is up arrow, add one row (if there is one available).
// The userData is the selectionRoot from main().
//
// Cheating, in a way, using a pointer into the structure -- ahh, doesn't
// matter.
//
void
myKeyPressCB(void *userData, SoEventCallback *eventCB)
{
    static int A=1, B=2; /* Rotation plane */
    static float r, f;
    const static float Rotation_Angle = 1.0;
    const static float Position_Step = 0.1;
    static int Shift_Down = ND_FALSE;

    const SoEvent *event = eventCB->getEvent();

    // check for the Up and Down arrow keys being pressed
    if (SO_KEY_PRESS_EVENT(event, UP_ARROW)) {
        printf("Up Key Event\n");

    } else if (SO_KEY_PRESS_EVENT(event, LEFT_SHIFT) ||
               SO_KEY_PRESS_EVENT(event, RIGHT_SHIFT)) {
        Shift_Down = ND_TRUE;

    } else if (SO_KEY_RELEASE_EVENT(event, LEFT_SHIFT) ||
               SO_KEY_RELEASE_EVENT(event, RIGHT_SHIFT)) {
        Shift_Down = ND_FALSE;

    } else if (SO_KEY_PRESS_EVENT(event, DOWN_ARROW)) {
        printf("Down Key Event\n");
        Draw_Klein_Bottle();

    } else if (SO_KEY_PRESS_EVENT(event, SPACE)) {
        printf("Pressed Space\n");
        ndMatrixMode(ND_WORLD_MODELVIEW);
        ndRotate (World_Dimension, A, B,
            (Shift_Down)? -Rotation_Angle : Rotation_Angle );
        Draw_Klein_Bottle();


    } else if (SO_KEY_PRESS_EVENT(event, I)) {
        ndMatrixMode(ND_WORLD_MODELVIEW);
```

```
            ndMatrixMode(ND_WORLD_MODELVIEW);
            ndLoadIdentity();
            Draw_Klein_Bottle();

        } else if (SO_KEY_PRESS_EVENT(event, P)) {
            Strip_Display = Strip_Display ^ 1;
            printf("Pressed 'p' - Strip_Display = %d\n",Strip_Display);
            Draw_Klein_Bottle();


        } else if (SO_KEY_PRESS_EVENT(event, S)) {
            printf("Pressed 's'\n");

        } else if (SO_KEY_PRESS_EVENT(event, NUMBER_1)) {
            A = 1;
            printf("Rotation Plane now: %d-%d\n", A, B);

        } else if (SO_KEY_PRESS_EVENT(event, NUMBER_2)) {
            A = 2;
            printf("Rotation Plane now: %d-%d\n", A, B);

        } else if (SO_KEY_PRESS_EVENT(event, NUMBER_3)) {
            A = 3;
            printf("Rotation Plane now: %d-%d\n", A, B);

        } else if (SO_KEY_PRESS_EVENT(event, NUMBER_4)) {
            A = 4;
            printf("Rotation Plane now: %d-%d\n", A, B);

        } else if (SO_KEY_PRESS_EVENT(event, A)) {
            B = 1;
            printf("Rotation Plane now: %d-%d\n", A, B);

        } else if (SO_KEY_PRESS_EVENT(event, B)) {
            B = 2;
            printf("Rotation Plane now: %d-%d\n", A, B);

        } else if (SO_KEY_PRESS_EVENT(event, C)) {
            B = 3;
            printf("Rotation Plane now: %d-%d\n", A, B);

        } else if (SO_KEY_PRESS_EVENT(event, D)) {
            B = 4;
            printf("Rotation Plane now: %d-%d\n", A, B);


        } else if (SO_KEY_PRESS_EVENT(event, Q)) {
            Destroy_Klein_Bottle();
            exit(0);

        } else {
            printf("Unhandled Key\n");
       }
    }


////////////////////////////////////////////////////////////////////////////////
//
// Klein Bottle Implmentation
////////////////////////////////////////////////////////////////////////////////
//
```

void Create_Klein_Bottle(float Radius)

```
{
    int   x, y;
    float t, a;
    float z = 2.0;
    float Angle_Increment = TWO_PI/Subdivisions;

    /* Allocate Main Array */
    Bottle = (Vector **) calloc(sizeof (Vector *), Subdivisions+1);
    CHECK_ALLOCATION(Bottle, "Create_Klein_Bottle() - Main Array");

    t = 0.0;
    for (x=0; x<=Subdivisions; x++)
```

```
    {
        Bottle[x] = (Vector *) calloc(sizeof(Vector *), Subdivisions+1);
        CHECK_ALLOCATION(Bottle[x], "Create_Klein_Bottle() - y loop");

        a = 0.0;
        for (y=0; y<=Subdivisions; y++)
        {
            Bottle[x][y] = (Vector) calloc(sizeof(Vector), 4);
            CHECK_ALLOCATION(Bottle[x][y], "Create_Klein_Bottle() - y loop");
            Bottle[x][y][0] = ( Radius + z * sin(a)) * cos(t);
            Bottle[x][y][1] = ( Radius + z * sin(a)) * sin(t);
            Bottle[x][y][2] = (z * cos(a)) * cos (t/2);
            Bottle[x][y][3] = (z * cos(a)) * sin (t/2);
            printf("Bottle[%d][%d] = (% 1.1f   %1.1f   %1.1f   %1.1f)\n",
                x, y,
                Bottle[x][y][0],Bottle[x][y][1],
                Bottle[x][y][2],Bottle[x][y][3]
            );
            a += Angle_Increment;
        }
        t += Angle_Increment;
    }
}
```

void Destroy_Klein_Bottle(void)

```
{
    int x,y;
    /* Deallocate each vertex */

    printf("Destroying Klein Bottle...\n");
    for (x=0; x<Subdivisions; x++)
    {
        for (y=0; y<Subdivisions; y++)
            /* Deallocate each vertex */
            free(Bottle[x][y]);

        /* Deallocate each row of vertices */
        free(Bottle[x]);
    }

    /* Deallocate main array */
    free(Bottle);
}
```

void Draw_Klein_Bottle(void)

```
{
    int x,y;

    Bottle_Coords->point.deleteValues(0);

    ndRenderMode(ND_FEEDBACK);

    for (x=0; x<=Subdivisions; x++)

        for (y=0; y<=Subdivisions; y++)
        {

            float Low_Dimensional_Vector[3];

            ndVertexNfv(4, Bottle[x][y]); // OK, but inverts on half
            ndGetfv(ND_RAW_DEVICE_VECTOR, Low_Dimensional_Vector);

            Bottle_Coords->point.setlValue (Bottle_Coords->point.getNum(),
             Low_Dimensional_Vector);  // Processed ND Vertex
        }
        if (x % 2 == 0 && Strip_Display) x+=2;  /* Draw every second strip */
    }

    myQuadMesh->verticesPerRow =
        (Strip_Display)? Subdivisions/2 : Subdivisions+1;
    myQuadMesh->verticesPerColumn =
        (Strip_Display)? Subdivisions/2 : Subdivisions+1;
}
```

```
/*****************************************************************************
 re: Understanding the definition of the Klein bottle.

 The following email from Mike Hartley at UWA may be of assistance.
 Many thanks Mike.

 ---------------------------------------------------------------------------
From hartley@maths.uwa.edu.au Mon Nov 13 11:38:54 1995
Received: from madvax.maths.uwa.edu.au by turing.fste.ac.cowan.edu.au (AIX 4.1/
UCB 5.64/4.03)
          id AA17340; Mon, 13 Nov 1995 11:38:51 +0800
Message-Id: <9511130338.AA17340@turing.fste.ac.cowan.edu.au>
Received: by madvax.maths.uwa.edu.au (5.61+IDA+MU)
id AA09909; Mon, 13 Nov 1995 11:38:21 +0800
From: hartley@maths.uwa.edu.au (Michael Hartley)
Subject: Re: Klein bottles again
To: aellertn@turing.fste.ac.cowan.edu.au (Andrew Ellerton)
Date: Mon, 13 Nov 95 11:38:19 WST
In-Reply-To: <9511100826.AA04390@turing.fste.ac.cowan.edu.au>; from "Andrew El-
lerton" at Nov 10, 95 4:26 pm
X-Mailer: ELM [version 2.3 PL0]
Status: OR

>
> Hi - Me again Mike!
>
> You guessed it, more 3 and 4 dimensional polytope-ish questions...
>
> I've been using your previous mail - the one with all the diagrams - as a
> basis for getting a neato little equation  to describe a point on the surface
> of a Klein bottle, but have a limited success.
>
> >From the way you built your discussion, I understand a 3D ring, and a ring of
> rings - a torus. A ring of radius r, centred at the origin, in the x-z plane is
> given by
>
> [1]           P = (r cos(theta), 0, r sin(theta))   : 0 <= Theta < 360
>
> No worries.
>
> A torus is constructed by sweeping a ring (or a band with width approaching 0)
> around a centre. For example, a torus swept about the y-axis, is given by,
> (to the best of my un-coxeter-educated mind) :
>
> [2]       p =  [ r cos(theta) + r cos(alpha)cos(theta)
>                   C             T
>
>                  r sin(alpha)
>                   T
>
>                  r sin(theta) + r cos(alpha)sin(theta)
>                   C             T                       ]
>
>           : 0 <= Theta < 360 and 0 <= Alpha <360
>
>           where
>                 r , r = inner and outer radius of torus respectively
>                  I   O
>
>                 r = radius of torus center to center of revolution
>                  C
>
>                   =(r + r  ) /2 + r
>                      I   O
>
>                 r = radius of the torus "tube" or ring
>                  T
>
>                   =(r - r ) / 2
>                      I   O
>
>
> Mmmm yes well, as I've said I haven't checked it.
```

Looks nice!!

>
> BUT thats not the problem - I figure the first step to building a Klein
> bottle is describing a mobius strip algebraically, and I can't do that, let
> alone the description of a general point on the surface of a Klein bottle.
>
> Can you offer any direction or assistance! (PLEASE?!)

Ooops! Sorry about the late reply....

Ok - Let's do a Mobius Band first...

suppose we want an ordinary band.. we could do it like this -

(x,y,z) = (r cos(t) , r sin(t) , 0) + (0, 0, z)

The first term is a circle, the next offsets the circle a little in the vertical direction. Now, let's put a twist in the band...

If we want our vertical offset to be not quite vertical, but tilted towards the centre by an angle a, then it ought to be

(z cos(t) sin(a) , z sin(t) sin(a) , z cos(a))

Now for a mobius band, we want a to go through a half turn (180d) in the "time" t m ; a full turn (360d). Thus, just let a=t/2, to get

((r + z sin(t/2))cos(t) , (r + z sin(t/2))sin(t) , z cos(t/2))

That should do the trick... if you make r=2 and z range from -1 to 1 it should look nice (and t ranges from 0 to 2Pi).

Now for a Klein Bottle... Actually, a Klein bottle can't be embedded in 3-D space without intersecting itself.. But that should be ok for your program, shouldn't it??

Let's have a go in 4-D.

This time, you take your circle (x,y,z,w) = (r cos(t), r sin(t), 0 , 0) and offset it by a circle (z sin(a) cos(t) , z sin(a) sin(t), z cos(a), 0), where instead of fixing a and making z the parameter, we fix z and make a the parameter. The simplest thing I can think of is this: slowly rotate the above in the z-w plane, so as to make a half turn by the time t has gone all the way from 0 to 2Pi. Thus,

```
[ x ]   [ ( r + z sin(a) ) cos(t) ]
[   ]   [                         ]
[ y ]   [ ( r + z sin(a) ) sin(t) ]
[   ] = [                         ]
[ z ]   [  ( z cos(a) ) cos(t/2)  ]
[   ]   [                         ]
[ w ]   [  ( z cos(a) ) sin(t/2)  ]
```

where r and z are constants, and t and a range from 0 to 2Pi.

And then make sure you give it some really strong perspective distortion before you view it...

I hope it works!!

> The principal reason I need it is for the demonstration of the n-dimensional
> viewing system (which I _hope_ I will finish!). Would you like to come? I
> believe the seminar is open to all. Its scheduled for the 24 november, time
> and place yet to be finalised. Whether or not I get Mr Klein Bottle working,
> you're more than welcome to come.

I think I'd be really interested... let me know more details, and hopefully I'll be free at that time...

I've recently finished a program that draws tesselations of 2-D hyperbolic space. It's written in turbo pascal..

Yours, Mike H...

```
                "If this sentence is true, then I am good at logic"
                         /
                        /+-----------------+
    . . . . .    __O| Michael Hartley +---------+          _-_|\
                -\<,| hartley@maths.uwa.edu.au |         /     \
  . . . . . .  ()/ ()+--------------------------+-------> *_.-._/
                                                          v

 ****************************************************************************/
```

B.9 Building Instructions

In comparison to the rest of the *ND* library archive, the inventor demonstration directory uses a straight-forward makefile for construction.

nd/progs/inventor/Makefile

```
# Default-ish Makefile - Very Default.
#
# Instructions:
# -------------
# None at this time.
#
# Andrew Ellerton # 960127

DIRECTORY_NAME = inventor

PROGRAMS = klein hcube
UNITS    =

include /usr/include/make/commondefs

INCLUDE = -I../../include
TARGETS = $(PROGRAMS)


#LLDLIBS = -lGLU -lX11 -lm -lInventorXt
LLDLIBS = -lGL -lGLU -lm -lInventorXt -L../../lib/nd -lND
C++FILES = $(PROGRAMS:=.c++)

default: $(PROGRAMS)

include $(COMMONRULES)
$(PROGPAMS): $$@.c++
$(C++F) $(INCLUDE) -o $@ $@.c++ $(LDFLAGS)

archive:
make clean
cd ..; tar -cvf - $(DIRECTORY_NAME) | gzip -c > $(DIRECTORY_NAME).tar.gz
```

Appendix C. Source Code: Test Drivers

Within this appendix the source code of each test drivers written for the *ND* library is provided. There are five test drivers; one for each of the main data structures used by *ND*, and: namely,

- matrices: `test_matrix.c`
- matrix stacks: `test_matrix_stack.c`
- vectors: `test_vector.c`

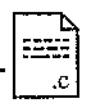
and one driver for two other areas of functionality: namely,

- rendering context "get" function: `test_get.c`
- matrix/vector multiplication: `test_mult.c`

Each test driver is essentially a main-line, with few additional functions. Within the main-line, the particular data structure or library functionality of interest is put through a series of tests to verify, to a degree of confidence, the completeness and correctness of that section of code. For example, the vector data structure needs to be able to create vectors of any size, free the memory of those vectors, multiply a vector by a matrix and so on.

Each driver, in alphabetical order, is listed below.

C.1 Test: Context "Get" Function



nd/lib/nd/test/test_get.c

```
/* This file should have only a main function which will test private and
   non private functions.

   Tested specifically in this file: ndGet.

   960124: Specifically for ndGet(RAW_DEVICE_VECTOR)
   960130: WARNING: This module incomplete. See nd/progs/inventor/klein.c++
   for example usage of ndGet

 */

#include <stdio.h>
#include <GL/nd.h>
#include "../private.h"
```

```
void Pause(void)
{
    printf("Press return to continue\n");
    getchar();
}
```

entry point:

```
int main(void)
{
    float *Float_Vector_Result;
    Vector V;
    float x = 99.1;
    int i;

    New_Vector(3, &V);

    printf("Dimension(V) should = 3, actually equals: %d\n", DIMENSION_OF(V));
    Put_Vector(V);
    printf("about to free_vector()... ");
    Pause();
    Free_Vector(V);
    Pause();

    /* ndGetfv(ND_RAW_DEVICE_VECTOR, Float_Vector_Result); */
    for (i=0; i<3; i++)
        printf("Raw Device Vector[%d] = % 1.3f\n", Float_Vector_Result[i]);

    V = Set_Vector(6, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6);
    printf("Dimension(V) should = 6, actually equals: %d\n", DIMENSION_OF(V));
    Put_Vector(V);

    getchar();

    Free_Vector(V);

    getchar();
    V = Set_Vector(12,  x-1.0, x-2.0, x-3.0, x-4.0,
                        x-5.0, x-6.0, x-7.0, x-8.0,
                        x-9.0, x-10.0, x-11.0, x-12.0);
    printf("Dimension(V) should = 12, actually equals: %d\n"
           "Contents of vector should decrease by 1.0\n", DIMENSION_OF(V));
    Put_Vector(V);
    Free_Vector(V);


    printf("Testing Complete\n");
    return 0;
}
```

C.2 Test: The Matrix Data Structure

nd/lib/nd/test/test_matrix.c

```
/* This file should have only a main function which will test private and
   non private functions.

   Tested specifically in this file: PRIVATE MATRIX FUNCTIONS
 */

#include <stdio.h>
#include <GL/nd.h>
#include "../private.h"
```

entry point:

```
int main(void)
{
    Matrix M, A, B, R;
    float x = 99.1;
    int i;
    double V[] = {1.1, 1.2, 1.3, 1.4, 1.5, 1.6};
    M = New_Matrix(4, 4);
```

```
    printf("Dimension(M) should = 4x4, actually equals: %d x %d\n",
           ROWS_OF(M), COLUMNS_OF(M));
    Put_Matrix(M);
    Free_Matrix(&M);
    Free_Matrix(&M);
    Free_Matrix(&M);
    Free_Matrix(&M);

    M = Set_Matrix(4, 4,
               x, 0.0, 0.0, 1.5,
               0.0, 1.0, 0.0, 2.5,
               0.0, x, 1.0, 3.5,
               0.0, 0.0, 0.0, 4.5);
    Put_Matrix(M);

    for (i=0; i<100; i++)
    {
    M = Set_Matrix(5, 5,
              11.0,12.0,13.0,14.0, 15.0,
              21.0,22.0,23.0,24.0, 25.0,
              31.0,32.0,33.0,34.0, 35.0,
              41.0,42.0,43.0,44.0, 45.0,
              51.0,52.0,53.0,54.0, 55.0);
    printf("Iteration %d\n", i);
    Free_Matrix(&M);
    }

    printf("\n\nIdentity:\n");
    M = Identity_Matrix(4);
    Put_Matrix(M);
    Free_Matrix(&M);

    printf("\n\nRotation: 4D, 2-4 Plane, 23.3 degrees:\n");
    M = Rotation_Matrix(4, 2, 4, 23.3);
    Put_Matrix(M);
    Free_Matrix(&M);

    printf("\n\nScale: 6D, [1.1, 1.2, 1.3, 1.4, 1.5, 1.6]:\n");
    M = Scale_Matrix(6, V);
    Put_Matrix(M);
    Free_Matrix(&M);

    printf("\n\nTranslation: 5D, [1.1, 1.2, 1.3, 1.4, 1.5]:\n");
    M = Translation_Matrix(5, V);
    Put_Matrix(M);
    Free_Matrix(&M);

    printf("\nTest matrix multiplication..."
           "\nR=A*B == A because B==Identity\n");
    A = Translation_Matrix(5, V);
    B = Identity_Matrix(6);
    MATRIX_ELEMENT(B,2,5)=923.2;
    Multiply_Matrices(&R, A,B);
    printf("\nMatrix A:\n");
    Put_Matrix(A);
    printf("\nMatrix B:\n");
    Put_Matrix(B);
    printf("\nResult Matrix:\n");
    Put_Matrix(R);
    Free_Matrix(&A);
    Free_Matrix(&B);
    Free_Matrix(&R);

    printf("\nTesting Complete.\n");
    return 0;
}
```

C.3 Test: The Matrix Stack Data Structure

nd/lib/nd/test/test_matrix_stack.c

```
/* This file should have only a main function which will test private and
```

```
    non private functio: .

    Tested specifically in this file: MATRIX AND MATRIX_STACK FUNCTIONS
 */

#include <stdio.h>
#include <GL/nd.h>
#include "../private.h"
```

entry point: **int main(void)**

```
{
    Matrix M;
    Matrix_Stack S;

    float x = 99.1;
    int i, j; /* Loop counters */
    const size_of_stack = 5;
    const Dimensions = 3;
    double V[20];


    M = New_Matrix(4, 4);
    S = New_Matrix_Stack(size_of_stack);

    for (i=1; i<size_of_stack; i++)
    {
        M = Set_Matrix(4, 4,
              0.0, 0.0, 0.0, (double) i,
              0.0, 0.0, 0.0, (double) i,
              0.0, 0.0, 0.0, (double) i,
              0.0, 0.0, 0.0, (double) i);
        printf("Iteration %d\n", i);
        Put_Matrix(M);
        printf("\n\n");
        Push_Matrix(S, M);
        Free_Matrix(&M);
    }

    printf("\n\nContents of the stack: \n");
    Put_Matrix_Stack(S);
    for (i=1; i<size_of_stack; i++)
    {
        printf("\n\nPrint and pop the top matrix: \n");
        Put_Matrix(*Pop_Matrix(S));
    }

    /* Should handle popping of an empty matrix */
    printf("\n\nShould get an error b/c of an empty stack: \n");
    (void) Pop_Matrix(S);

    printf("\n\nShould get an error dealloc'ing stack b/c its empty: \n");
    Free_Matrix_Stack(S);

    /* SECTION 2 */
    printf("\n\nSECTION 2: Trying with %d-Dimensionsal stack: \n", Dimensions);

    S = New_Matrix_Stack(size_of_stack);
    for (i=0; i<size_of_stack; i++)
    {
        for (j=0; j< Dimensions; j++)
            V[j] = (double) i;
        M = Translation_Matrix(Dimensions, V);
        printf("Iteration %d\n", i);
        Put_Matrix(M);
        printf("\n\n");
        Push_Matrix(S, M);
        Free_Matrix(&M);
    }

    printf("\n\nContents of the stack: \n");
    Put_Matrix_Stack(S);

    printf("\n\nShould be NO PROBLEMS deallocing the stack, %d elements: \n",
        size_of_stack);
    Free_Matrix_Stack(S);
```

```
    return 0;
}
```

C.4 Test: Matrix/Vector Multiplication

nd/lib/nd/test/test_mult.c

```
/* This file should have only a main function which will test private and
   non private functions.

  Tested specifically in this file: PRIVATE MATRIX/VECTOR MULTIPLICATION FUNC-
TIONS
 */

#include <stdio.h>
#include <GL/nd.h>
#include "../private.h"
```

entry point: **int main(void)**

```
{
    Matrix M, A, B, R;
    float x = 99.1;
    int i, D;
    double V[] = {3.1, 3.2, 3.3, 3.4, 3.5, 3.6, 3.7, 3.8, 3.9, 3.10};
    M = New_Matrix(4, 4);

    printf("Dimension(M) should = 4x4, actually equals: %d x %d\n",
            ROWS_OF(M), COLUMNS_OF(M));
    Put_Matrix(M);
    Free_Matrix(&M);

    M = Set_Matrix(4, 4,
               x, 0.0, 0.0, 1.5,
               0.0, 1.0, 0.0, 2.5,
               0.0, x, 1.0, 3.5,
               0.0, 0.0, 0.0, 4.5);
    Put_Matrix(M);


    printf("\n\nIdentity:\n");
    M = Identity_Matrix(4);
    Put_Matrix(M);
    Free_Matrix(&M);

    R = Identity_Matrix(3);

    printf("\nTest matrix multiplication..."
           "\nR=A*B == A because B==Identity\n");
    A = Translation_Matrix(5, 7);
    B = Identity_Matrix(6);
    Multiply_Matrices(&R, A,B);
    printf("\nMatrix A:\n");
    Put_Matrix(A);
    printf("\nMatrix B:\n");
    Put_Matrix(B);
    printf("\nResult Matrix:\n");
    Put_Matrix(R);
    Free_Matrix(&A);
    Free_Matrix(&B);

    Free_Matrix(&R);

    for (i=1; i<20; i++)
    {
        printf("\nMultiplication Iteration %d\n",i);
        D = Random_Int(3, 10);
        printf("\nRandom number = %d\n", D);

        printf("\nTest matrix multiplication: two %d x %d matrices."
               "\nR=A*B\n", D+1, D+1);
        A = Rotation_Matrix(D, 1, 2, 23.3);
```

```
        B = Translation_Matrix(D, V);
        Multiply_Matrices(&R, A,B);

        printf("\nMatrix A:\n");
        Put_Matrix(A);
        printf("\nMatrix B:\n");
        Put_Matrix(B);
        printf("\nResult Matrix:\n");
        Put_Matrix(R);

        Free_Matrix(&A);
        Free_Matrix(&B);
        if (i < 10) Free_Matrix(&R);
        if (i < 5) Free_Matrix(&R);
    }

    printf("\nMATRIX MULTIPLICATION Testing Complete.\n\n");

    printf("\nVECTOR & MATRIX multiplication..."
           "\nR=A*V ...\n");
    A = Translation_Matrix(3, V);
    V = New_Vector(4);
    Multiply_Matrix_By_Vector(&RV, A,V);
    printf("\nMatrix A:\n");
    Put_Matrix(A);
    printf("\nVector V:\n");
    Put_Vector(V);
    printf("\nRESULT Vector:\n");
    Put_Vector(RV);

    printf("\nTesting Complete.\n");
    return 0;
}
```

C.5 Test: The Vector Data Structure

nd/lib/nd/test/vector.c

```
/* nd/lib/nd/test/vector.c

   This file should have only a main function which will test private and
   non private functions.

   Tested specifically in this file: PRIVATE VECTOR FUNCTIONS
 */

#include <stdio.h>
#include <GL/nd.h>
#include "../private.h"

void Pause(void)
{
    printf("Press return to continue\n");
    getchar();
}
```

entry point:

```
int main(void)
{
    Matrix M;
    Vector V;
    float x = 99.1;
    int i;

    M = New_Matrix(4, 4);
    New_Vector(3, &V);

    printf("Dimension(V) should = 3, actually equals: %d\n", DIMENSION_OF(V),;
    Put_Vector(V);
    printf("about to free_vector()... ");
    Pause();
    Free_Vector(V);
```

```
    Pause();


    V = Set_Vector(6, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6);
    printf("Dimension(V) should = 6, actually equals: %d\n", DIMENSION_OF(V));
    Put_Vector(V);

    getchar();

    Free_Vector(V);

    getchar();
    V = Set_Vector(12,  x-1.0, x-2.0, x-3.0, x-4.0,
                        x-5.0, x-6.0, x-7.0, x-8.0,
                        x-9.0, x-10.0, x-11.0, x-12.0);
    printf("Dimension(V) should = 12, actually equals: %d\n"
           "Contents of vector should decrease by 1.0\n", DIMENSION_OF(V));
    Put_Vector(V);
    Free_Vector(V);

    printf("\nTest set/free vector, 100 iterations\n");
    for (i=0; i<100; i++)
    {
        V = Set_Vector(5, 11.0,12.0,13.0,14.0, 15.0);
        printf("Iteration %d\n", i);
        Free_Vector(V);
    }

    printf("\nTest new/free vector, 100 iterations."
           "\n(Also tests error handling because size <= 0\n");
    for (i=0; i<100; i++)
    {
        New_Vector(i-5, &V);
        printf("Iteration %d: Size of vector = %d\n", i, i-5);
        Free_Vector(V);
    }

    printf("\nTest copy vector, 30 iterations."
           "\n(Each vector is i dimensions, (i+1.0, i+2.0 ... i+i)\n");
    for (i=0; i<30; i++)
    {
        Vector C;
        int j;

        New_Vector(i, &V);
        for (j=1; j<=i; j++)
            VECTOR_ELEMENT(V,j) = (Element_Type) i+j;
        C = Copy_Vector(V);

        printf("\nIteration %d: Size of vector = %d\n", i, DIMENSION_OF(V));
        printf("\nOriginal vector:\n");
        Put_Vector(V);
        printf("\nDuplicate vector:\n");
        Put_Vector(C);

        Free_Vector(V);
        Free_Vector(C);
    }

    printf("Testing Complete\n");
    return 0;
}
```

C.6 Building the Test Drivers

In order to build each of the drivers into an executable program, the dependencies of each unit must be taken into consideration and an appropriate compilation carried out. As with the ND library and demonstration programs, the imake command, with an appropriate

Imakefile, has been used to construct a platform-independent makefile, that in-turn performs compilation as required. The Imakefile for the test-drivers is listed below.

nd/lib/nd/test/Imakefile

```
/* Based on the Imake file by Mark J. Kilgard (for GLUT)
 *
 * A.Ellerton 1995
 */

/* NO subdirs */

#include <Library.tmpl>

#include "../../../ND.cf"

TARGETS = test_matrix  test_matrix_stack    test_vector   test_mult


AllTarget($(TARGETS))

 ND_DIR =../

ComponentTestTarget(test_matrix, $(ND_DIR)matrix.o $(ND_DIR)vector.o
    $(ND_DIR)transformations.o $(ND_DIR)context.o $(ND_DIR)matrix_stack.o )
ComponentTestTarget(test_get,   $(ND_DIR)matrix.o $(ND_DIR)vector.o
    $(ND_DIR)context.o $(ND_DIR)vertex.o $(ND_DIR)matrix_stack.o
    $(ND_DIR)transformations.o )
ComponentTestTarget(test_vector,        $(ND_DIR)matrix.o $(ND_DIR)vector.o )
ComponentTestTarget(test_matrix_stack,  $(ND_DIR)matrix.o $(ND_DIR)vector.o
    $(ND_DIR)matrix_stack.o $(ND_DIR)transformations.o $(ND_DIR)context.o)

###ComponentTestTarget(test_mult,          $(ND_DIR)matrix.o $(ND_DIR)vector.o
    $(ND_DIR)transformations.o)

DependTarget()
```